University of Zurich[UZH]

Implementing Support for SPARQL Filters in TripleRush

Bachelor Thesis January 30, 2015

Lucas Jacques
of Thalwil ZH, Switzerland

Student-ID: 11-756-228
jacqueslk@gmail.com

Advisor: **Bibek Paudel**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor, Bibek Paudel, who has guided me throughout the course of this thesis with helpful suggestions and comments and who kindly introduced me to various technical subjects. Our weekly meeting kept me motivated and provided me with continuous feedback.

I would also like to thank the Dynamic and Distributed Information Systems Group of the University of Zürich, and especially its head, Prof. Dr. Abraham Bernstein, for giving me the unique opportunity to write this thesis and for granting me access to the university's cluster to test and evaluate my additions to TripleRush.

Finally, I would like to thank my parents for their support throughout my studies.

Zusammenfassung

Das Semantische Web – ein Netz von Daten, welches aus verknüpften RDF-Daten gebildet wird – wächst stets. RDF-Daten können in sogenannte Triplestores verwaltet werden, welche ebenfalls Unterstützung für SPARQL (SPARQL Protocol and RDF Query Language) bieten. Mithilfe von SPARQL können RDF-Daten abgerufen werden, welche benutzerdefinierten Bedingungen entsprechen.

TripleRush ist ein solches Triplestore und benützt eine graph-basierte Struktur, um SPARQL-Abfragen effizient beantworten zu können. Diese Arbeit beschreibt die Erweiterungen in TripleRush zur Unterstützung von SPARQL-Filtern. In dieser Arbeit wird erläutert, wie die Filter in TripleRush intern dargestellt werden und auf welche Weise die Filter während dem Ausführen von SPARQL-Abfragen überprüft werden.

Abstract

The Semantic Web – a web of data formed by interlinked RDF data – has seen a steady increase in size. Triple stores are data management systems for RDF data and offer support for the SPARQL Protocol and RDF Query Language (SPARQL). With SPARQL, RDF data can be retrieved which satisfy user-defined criteria.

TripleRush is such a triple store, using a graph-based architecture to efficiently answer SPARQL queries. This thesis discusses the implementation of SPARQL filter support in TripleRush. We discuss how the filters are represented after they have been parsed and describe how they are checked during query execution.

Table of Contents

1

Introduction

Parallel to the document web there exists the *Semantic Web*, a graph of data, formed by data from different datasets connected together by means of references among each other. The Semantic Web has seen a steady increase in size; DBpedia, for instance, contains 3 billion facts[1]. In contrast to the traditional document web, which is designed for humans, the Semantic Web centers around machine-readable data, such that programs can process and combine data from different datasets in one generic manner. Thanks to this uniformity, developers are not burdened with the cumbersome task of converting between different structures and data formats to join data from multiple sources.

The Semantic Web consists of *Linked Data* – data adhering to a set of publishing principles. In Linked Data, all data are expressed as *RDF triples*, which consist of a subject, a predicate and an object. Each triple states a fact about the subject; it has a relationship with the object that is described by the predicate. For example, we could state that Switzerland lies in Europe with the triple `(Switzerland, continent, Europe)`. An *RDF graph* can be formed from the RDF triples, wherein the subjects and objects act as vertices, and the predicates connect the vertices as directed, labeled edges.

The *SPARQL Query Language* is the standard querying language for RDF graphs. It allows the retrieval of values from RDF graphs which satisfy user-defined queries and constraints. For instance, a list of European countries with more than 10 million inhabitants could be generated by querying geopolitical datasets. The results of a SPARQL query are not returned as a graph but as rows, where each row specifies a combination of values which satisfy the query, akin to SQL.

Collections of RDF triples are saved in *triple stores*, which facilitate the management of triples and allow the retrieval of data with SPARQL queries (i.e. criteria supplied by the user), similar to data management tools for relational databases.

TripleRush is such a triple store. The aim of TripleRush is to efficiently answer SPARQL queries over large collections of RDF triples. It is built on top of the graph processing framework Signal/Collect. Prior to this thesis, TripleRush lacked support for SPARQL filters, which are comparable to the `WHERE` clause in SQL. This thesis discusses the implementation of filter support, detailing how the filter information is parsed and incorporated into the process of query execution.

[1] Year: 2014, *http://blog.dbpedia.org/category/dataset-releases/*

1.1 Contributions

This thesis discusses the implementation of SPARQL filter support in TripleRush, from the process of parsing the filters in the beginning to the subsequent evaluation of the filters, such that the returned results of SPARQL queries also satisfy the constraints specified in the filters. Furthermore, this thesis sheds some light on the structure of TripleRush, particularly about the special graph – the *index graph* – it employs and about the manner in which the query information is passed among its nodes.

1.2 Document Structure

The structure of this thesis is as follows. Chapter 2 provides an overview of the Linked Data principles and technologies and presents various implementations. Chapter 3 describes the architecture of TripleRush in detail. Chapter 4 presents the implementation of filter support in TripleRush. In Chapter 5, we discuss the evaluation results of TripleRush for queries with filters, before finishing with a conclusion in Chapter 6.

2

Related Work

In this chapter, we provide a background in the standards of the Semantic Web and discuss the approaches employed by various triple stores to implement the Linked Data standards.

2.1 Linked Data

Traditionally, data on the internet have been published as HTML documents, raw dumps, or made accessible via APIs. These approaches, however, render it difficult to combine data from multiple datasets and therefore, they remain largely isolated from each other [Heath and Bizer, 2011].

HTML (HyperText Markup Language) is "the publishing language of the World Wide Web," [Raggett et al., 1999] providing a means to format documents on the web, such as adding emphasis to text, embedding images or creating tables. The World Wide Web Consortium (W3C) goes as far as to refer to HTML as the core language of the World Wide Web in the latest HTML specification, HTML5 [Hickson et al., 2014].

However, the purpose of HTML is to structure text on a webpage – not data. Someone wishing to use the data displayed on a web page is thus faced with the difficult task of extracting the data from the HTML page, which are often mixed with other text.

Data can be easily accessed via an API or a data dump (e.g. data dumped into a single XML or CSV file). While these access methods do not require any filtering between data and other text, there is no standard way of integrating multiple datasets as they may employ different formats and likely do not use the same identifiers for their entries. Additionally, APIs may differ in the methods that they offer to the user.

Linked Data alleviates these problems by following a set of principles for publishing data. With data abiding to the Linked Data principles, not only is one be able to combine data from different datasets in a generic way, but it should also be possible for any data publisher to add facts about a subject on the web and to refer to other datasets, just as anyone may publish a document on the document web and link to other documents.

Similar to the document web, where new documents can be found by following links to other webpages, new data should be discoverable by following references to other datasets. This creates a web of data – the so-called *Semantic Web* – which can be

explored by hopping from dataset to dataset by continuously following external links (i.e. links from one dataset to others).

The *Resource Description Framework* (RDF) is a family of formats with which Linked Data can be published. Every fact is expressed as an *RDF triple*, which consists of a subject, a predicate and an object. The subject is the topic one wishes to state a fact about and can be any topic at all, such as a book, a certain event or a programming language. The predicate describes the relationship between the subject and the object.

Subject	Predicate	Object
Switzerland	continent	Europe
Switzerland	population	8183000
Switzerland	memberOf	United Nations
Nepal	continent	Asia
Nepal	population	26495000
Nepal	memberOf	United Nations

Table 2.1: Sample triples

Consider the triples given in Table 2.1. The first triple states that Switzerland, the triple's subject, has a relationship with the object, `Europe`. The predicate, `continent`, describes the type of the relationship, i.e. it specifies that Europe is the continent in which Switzerland lies.

An *RDF graph* can be formed from a collection of triples by using the subjects and objects as vertices, and by regarding each predicate as a directed, labelled edge from the subject to the object of the according triple. Figure 2.1 shows the RDF graph resulting from the sample triples.

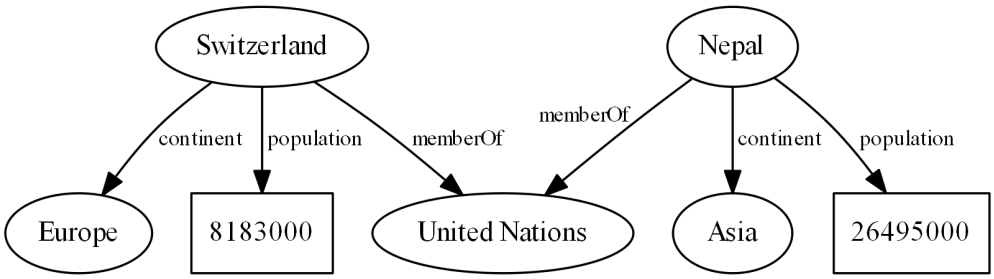

Figure 2.1: RDF graph resulting from the sample triples

We note that it is possible for two vertices to be connected with multiple edges. For instance, if we added the triple `(Switzerland, likes, Europe)` to the sample triples in Table 2.1, the vertex `Switzerland` would be connected to `Europe` with two edges, labelled `continent` and `likes`, respectively.

For identification, a unique Uniform Resource Identifier (URI) is assigned to each subject and predicate, while the object may be either a URI or a literal value. In other words, all entities – so-called *resources* – RDF triples refer to are identified by URIs. In

addition, HTTP URIs should be chosen for the resources [Berners-Lee, 2006]. In this manner, a URI can be easily looked up over HTTP and the page it returns should contain additional triples about the resource. This has the advantage that no new protocols must be adopted for Linked Data, i.e. it builds on top of the existing architecture of the web [Bizer et al., 2009]. The best practices for defining and handling URIs in Linked Data are discussed in detail in [Ayers and Völkel, 2008].

In the RDF graph given in Figure 2.1, all round vertices would be identified by URIs, whereas the square vertices `8183000` and `26495000` would be literals, i.e. the object field would hold the specific value in the RDF triple – not a URI. Discovery of additional triples is possible by dereferencing any of the URIs, which may point to the same dataset or to an external one. Furthermore, ontologies are provided to allow publishers to state that a URI is equivalent to another one[1]. Essentially, individual datasets are connected by such equivalency statements and by the use of external URIs as objects in RDF triples, thus forming the Semantic Web.

2.2 SPARQL

The *SPARQL Protocol and RDF Query Language* (SPARQL) is a language for querying and modifying RDF graphs, comparable to how SQL is used to retrieve and manipulate data in relational databases. SPARQL allows the retrieval of data which satisfies semantic criteria given in a query. Results are returned as rows, where each row specifies a possible combination of values for all variables in the `SELECT` statement, as it is done in SQL. Consider the following SPARQL query:

```
SELECT ?country ?pop
WHERE {
    ?country <continent> <Europe> .
    ?country <population> ?pop
}
```

The fields for which values should be returned are declared as variables, which are denoted by a starting question mark, such as `?country`. The `SELECT` statement specifies that we want to retrieve the values for the variables `country` and `pop`. The criteria the values must meet is supplied in the `WHERE` clause in form of *triple patterns*. As in RDF, each statement consists of three fields – subject, predicate and object – with the difference that any field in a triple pattern may be a variable.

In the query above, given a resource `X`, we state that we only want `X` returned as a result for the variable `country` if there exists a triple `(X, continent, Europe)`. If this is the case, we ensure that a pattern `(X, population, Y)` exists for the same `X` and some value `Y`, whereupon we can return the row `(X, Y)` as a result for `?country` and `?pop`. This process is repeated for every possible combination of `X` and `Y`; each combination is represented as a row. All rows constitute the result of the query.

[1] Most notably, the OWL Web Ontology Language, *http://www.w3.org/TR/owl2-overview/*

Given the sample triples in Table 2.1, the query only has one result, namely Switzerland and its population. The country Nepal does not satisfy the query as there is no triple `(Nepal, continent, Europe)`.

To further refine queries, it is also possible to supply *filters* along with triple patterns. Filters express additional constraints on the *binding* of a variable, i.e. filters define criteria which must be satisfied by the value assigned to a variable, or the binding will be discarded. The following query contains such a filter:

```
SELECT ?country
WHERE {
    ?country <population> ?pop
    FILTER(?pop >= 10000000)
}
```

In contrast to the previous query, we declare in `SELECT` that we only wish to retrieve the values for the variable `country`. The query contains one triple pattern with two variables: `(?country <population> ?pop)`. Based on this triple pattern, we assign every possible subject `S` and object `O` for which a triple `(S, population, O)` exists to the variables `country` and `pop`. Additionally, we require in the filter that the value bound to `?pop` be at least 10,000,000.

Given the triples in Table 2.1, the only result is Nepal, as the population of Switzerland is too small to satisfy the filter. During query execution, we may try to assign `Switzerland` and its population to `?country` and `?pop`, respectively, but the filter will fail since the value of `?pop` is less than 10,000,000 in this case.

2.3 Graph Database Systems

Graph databases have proven to be useful in various domains, such as in the transport or communications sector. For instance, Twitter keeps track of who follows whom in a graph database and queries it to suggest new accounts to follow [Gupta et al., 2013]. Since a collection of RDF triples forms a graph, it is possible to use general-purpose graph engines to execute queries over RDF graphs. We now discuss a couple of real-world systems.

Pregel [Malewicz et al., 2010] is a distributed graph processing engine, designed to support graphs of large size. The vertices of the graph execute computations in Pregel and possess a state, which may be *active* or *inactive*. Prior to the execution of an algorithm, all vertices have the state *active*. Results are computed in an iterative manner; all active vertices execute a user-defined function in each iteration, during which they can send messages along their edges, change the graph topology, and they can choose to switch their status to *inactive*. Inactive vertices are only reactivated if they receive a message. The execution of the algorithm is terminated when all vertices are inactive.

The graph is divided into partitions and distributed among the machines of the cluster when it is in a distributed setting. The ID of a vertex is the only parameter used

to determine to which machine it will belong and by default, the assignment function `hash(ID) mod N` is used, where `N` designates the number of available machines. However, Pregel offers the possibility to the user to replace the assignment function with a custom one. Indeed, the authors note that performance can be enhanced for the Web graph if all vertices representing documents of the same webpage are kept together. [Malewicz et al., 2010]

In contrast, the **PowerGraph** abstraction [Gonzalez et al., 2012] employs a more elaborate distribution strategy. The authors argue that graphs representing real-world data usually have power-law degree distributions. In other words, they assume that graphs are typically sparse but that a small fraction of the vertices have a high degree, i.e. many vertices in the graph have few neighbors and only few have many.

Pregel distributes a graph over a cluster by assigning each vertex to a machine. Therefore, sending messages over an edge whose vertices are not on the same machine incurs network overhead. PowerGraph, in turn, uses *vertex-cuts* to separate a graph: different portions of the same vertex may reside on several machines. For every edge in the graph, its vertices are located on the same machine, such that sending and receiving messages over the edges does not require any communication over the network.

However, as certain vertices are divided into multiple pieces, synchronization becomes necessary. One piece of each vertex acts as the master, to which all other pieces send an aggregated version of the messages they have received. The master may update its status based on the incoming information and sends a reply to every piece to inform them of the new status. From there on, each piece copies the status and may send messages to its outgoing edges to inform the neighbors of the change. Due to the vertex-cut, sending messages over the edges of any piece does not require any communication over the network. Thus, only the synchronization between the pieces takes place over the network, which consists of two messages for each piece per update.

2.4 Triple Stores

Contrary to the graph systems illustrated in Section 2.3, triple stores offer native support for SPARQL as they are specifically built for the management of RDF triples. Typically, they are designed with efficiency for SPARQL queries in mind.

A handful of commercial systems exist which are specialized in the management of RDF data, such as Jena [McBride, 2001], Sesame [Broekstra et al., 2002] and 4Store [Harris et al., 2009]. Virtuoso – a commercial, general purpose database engine – has also been extended to support RDF storage and querying by translating SPARQL queries internally to SQL [Erling and Mikhailov, 2009].

We now discuss further triple stores and the approaches they use to evaluate queries. **TriAD** [Gurajada et al., 2014] is a distributed, shared-nothing RDF engine which uses asynchronous messages to join partial bindings. One node is assigned as the master node, which holds metadata about the stored RDF triples and is responsible for the coordination of all the other nodes, the so-called slave nodes.

Each slave node disposes of a disjoint part of the RDF data, connected with each other

by the predicates of triples whose subject and object are not stored in the same part of the data. A node operates over its part of the data and passes a message to another node if it must take over the operation because it disposes of the necessary data.

RDF-3X [Neumann and Weikum, 2009] stores all RDF data in one big table, rather than maintaining a table for each predicate, for example. A clustered B+ tree is created for every possible permutation of the triple fields from the big table, i.e. a B+ tree with the fields SPO, SOP, PSO, POS, OSP and OPS is generated, where S, P and O stand for subject, predicate and object, respectively.

Instead of storing the literal values inside the trees, they are encoded as numbers and can be looked up in a mapping index to return the real values. By keeping the IDs for the literal values in lexicographical order, it is easy to look up a certain range in any of the B+ tree, thereby avoiding random accesses. [Neumann and Weikum, 2009] states that the size of the six B+ trees is typically smaller than the original data thanks to the encoding.

Microsoft's **Trinity.RDF** [Zeng et al., 2013] uses a distributed, memory-based graph engine and stores RDF data in their graph form instead of using a relational database for storage. Thanks to the graph form, it avoids costly join operations and supports queries over graphs which cannot be expressed in SPARQL, such as determining the shortest path between two vertices.

It builds on top of *Trinity*, a graph engine whose goal is to support fast random accesses and efficient processing of large graphs. Rather than computing all bindings for each triple pattern separately and then joining these together, Trinity.RDF uses graph exploration to evaluate queries. As discussed in Section 2.1, RDF triples form a graph, in which the predicates act as labelled, directed edges between the vertices, i.e. the subjects and objects. For example, to evaluate a query with the triple pattern `(?country <continent> <Europe>)`, Trinity.RDF searches for all vertices which have an outgoing edge of type `continent` to the vertex `Europe`. For all additional triple patterns, it continues to search for the existence of edges and vertices until all triple patterns are satisfied.

2.5 Database Indexing

Database systems store data as blocks in a storage system. Additionally, they maintain various indices for one or more fields of a table. In this manner, look-up time can be improved as it is not necessary to iterate through all of the table's data if only a portion of each row is required. Indices serve as "shortcuts" to table rows: they duplicate the values of one or more fields of a table and store them as ordered entries. For each entry, a pointer to the physical location of the full row in the storage system is saved alongside.

For instance, if we store the population of countries into a table with two rows, `country`, as primary key, and `population`, we may decide to build an index for the `population` field. Then, if we want to select all countries which have between 10,000,000 and 20,000,000 citizens, we can select all entries in that range from the index and follow the pointers to the actual table rows in order to return the country names as results.

Without such an index, we would have to go through all rows in the table to find the results, since it is not ordered by population in the storage system.

Indices are divided into *primary* and *secondary* indices and can be described as *sparse* or *dense*. Primary indices contain the primary key and are therefore guaranteed not to have any duplicate entries. Furthermore, the entries of a primary index are stored in the same order as the table's data [Hector et al., 2009]. Secondary indices are indices for other fields and therefore do not have such guarantees.

Dense indices contain an entry for every value of a field, and save a pointer to where the entire table row is stored physically for each entry. In contrast, sparse indices only contain a portion of the values which are stored in the database. Maintaining a sparse index for a field is therefore only useful if it is a primary index, in which case all rows for values which are not present in the index are known to reside between the pointers of the closest entries in the index.

Since indices mirror a portion of a table and are accessed during query execution, it is important that their structure be easy to update and that it support fast searches. Indices are most often implemented with B+ trees [Hector et al., 2009], which are ordered and *self-balancing*, i.e. the distance from the root to any leaf node is the same in the entire tree. For larger secondary indices, hash tables may be used instead, as data are easily distributed with a hash function, whereas insertion into a B+ tree is more complex, as to ensure that the tree remains balanced.

Contrary to graph database systems, relational databases have received a lot of attention for several decades, and various indexing and optimization strategies have been developed, such as in [Selinger et al., 1979], which presents heuristics to best optimize the execution of a SQL query with the help of indices. [Chaudhuri and Narasayya, 1997] describes a system which proposes what fields an index should be built for, based on a collection of SQL queries provided by the user.

Newer research has gravitated towards allowing databases to automatically adapt to the data they manage and the queries they execute. *Adaptive indexing* refers to the process of automatically creating and refining indices based on the queries which are being processed [Graefe et al., 2011]. Challenges include finding algorithms generic enough to be applied on various types of data, and identifying how fast the database should react to changes in queries, since computing indices incurs cost.

Cracked databases, on the other hand, continuously reorganize the physical storage of the data to enable faster, more precise retrieval of relevant data [Idreos et al., 2007]. A column's data are *cracked* – physically reorganized into various partitions – each time they are processed by a query, separating the data which satisfy the query's constraints from those which do not. A so-called navigational map has to be maintained because the data are repeatedly cracked. As future queries require data from the cracked row, they refer to the map to locate the relevant portions of the data and thereby avoid the need to search through the data themselves.

3

TripleRush

We now take a detailed look at TripleRush [Stutz et al., 2013], a parallel in-memory triple store written in the Scala programming language. Built on top of the graph processing framework Signal/Collect [Stutz et al., 2010], it uses message-passing within its novel, graph-based architecture to execute SPARQL queries.

At the heart of TripleRush there lies a special graph based on the RDF triples it stores, called the *index graph*. To evaluate a query, a so-called *query particle* is constructed, explained in detail in Section 3.4. The query particle stores the triple patterns of the SPARQL query and adds bindings for the variables of the query as it is sent among the vertices of the index graph. Multiple copies of the query particle are created each time there are several values a variable can be bound to, allowing every new query particle to store a different binding and to pursue its own route independently.

The query particles are sent as messages from one vertex to another. Messages can be passed in parallel and asynchronously. Thanks to these properties, central bottlenecks caused by slow partial computations are avoided and first results can be reported as soon as they arrive, instead of having to wait until the query execution has terminated [Stutz et al., 2013].

3.1 Signal/Collect

Signal/Collect [Stutz et al., 2010] is a programming model for executing graph algorithms in a distributed setting. The vertices of the graph are the computing units, exchanging information by sending signals among each other. Afterwards, the vertices collect all incoming messages and handle them in a user-defined manner, such as by performing a computation based on the received messages. Algorithms are expressed as iterations of these two phases, namely signaling and collecting, from which the Signal/Collect model takes its name.

Algorithms may be executed in synchronous or asynchronous mode. In synchronous mode, a global synchronization point between the signaling and collecting phase guarantees that the vertices only enter the collecting phase once all have completed their signaling operation. In asynchronous mode, however, vertices call the signal or collect function in random order and independently of the other vertices. In this manner, the entire execution of the algorithm does not have to wait on slower partial computations.

Moreover, Signal/Collect includes various other functions to offer as much flexibility as possible. In particular, it supports the definition of several vertex and edge types for differentiation and additional attributes may be added to them. An aggregation vertex can be introduced to the graph to easily aggregate information from a given set of vertices. Furthermore, both in synchronous and asynchronous mode, it is possible to prioritize signal/collect operations by assigning a number of importance to them. A minimum threshold of importance may be set, below which operations are not executed. This provides an additional way for an algorithm to end: when all operations do not meet the minimum threshold, there is no longer anything to execute.

3.2 The Index Graph

TripleRush is a triple store built on top of Signal/Collect. It stores RDF triples in form of an *index graph*, a novel architecture designed to efficiently answer SPARQL queries over a large collection of data. The index graph is built by replacing one or two fields of each RDF triple with a wildcard, represented by `*`. Every possible combination of wildcards is used as an index and is added to the index graph, i.e. a triple `(S, P, O)` produces the indices `(S, *, *)`, `(*, P, *)`, `(*, *, O)`, `(S, P, *)`, `(S, *, O)`, and `(*, P, O)`. Additionally, the index graph contains a root node, `(*, *, *)`.

In other words, to include the triple `(Switzerland, continent, Europe)`, we add the following indices to the index graph: `(Switzerland, *, *)`, `(*, continent, *)`, `(*, *, Europe)`, `(Switzerland, continent, *)`, `(Switzerland, *, Europe)`, and `(*, continent, Europe)`.

We distinguish different index types by the information they store, i.e. by the fields which are not wildcards. For example, the index `(*, continent, *)` only contains a concrete value for the predicate and is therefore referred to as a P-index. Similarly, `(Switzerland, *, Europe)` is called an SO-index as it contains information about the subject and the object.

To reduce the necessary storage space of the index graph, all values are dictionary encoded: instead of storing the actual values of the RDF triples in each index, they are stored in an external dictionary where a unique, positive integer is assigned to each entry. The indices of the index graph only save the numbers of the according entries in lieu of the actual values, which saves a lot of space.

All indices have children of one other index type. A child of an index contains the same values in all fields which are already present in the parent. Moreover, one additional field of the child has a value where the parent has a wildcard. For example, P-indices have SP-indices as children. The structure of the index graph is given in Figure 3.1. For instance, the root index `(*, *, *)` might have a child `(*, continent, *)`, whose children may be the SP-indices `(Switzerland, continent, *)` and `(Nepal, continent, *)`.

The indices are connected to form the index graph by such parent-child relationships, acting as directed edges. Since every index type only has children of another index type, we only have to store the additional value of each child – the so-called *child delta* – in a list to reconstruct the children's fields.

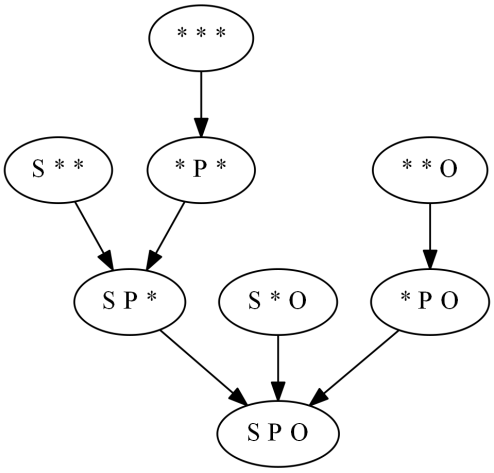

Figure 3.1: The index graph structure

For instance, provided that `(*, continent, *)` has `(Switzerland, continent, *)` and `(Nepal, continent, *)` as children, it is sufficient to store the values `Switzerland` and `Nepal` in the P-index to know all fields of its children. This requires little space, especially since we only need to save the dictionary ID of the values.

We note that an index may have multiple parents, e.g. both S-indices and P-indices have SP-indices as children. In an attempt to keep the number of children per index as little as possible, predicates are used as field for the child delta wherever possible. RDF datasets typically contain far less predicates than subjects or objects, which renders the predicate field ideal for child deltas.

Considering that SO-indices contain all possible predicates between their subject and object as child deltas, we realize that the SPO-indices are redundant as the SO-indices can reconstruct all SPO-indices. Therefore, SPO-indices are omitted in the index graph, contrary to the structure given in Figure 3.1. Oftentimes, a query has to be routed to an SPO-index to verify that bound variables satisfy another triple pattern in the query. TripleRush reroutes such checks to the corresponding SO-index by replacing the predicate with a wildcard. The index then verifies whether or not an RDF triple exists with the given subject, predicate and object by searching for the predicate in its list of child deltas.

3.3 Technical Layout of the Index Graph

We now take a look at the technical layout of the indices and their most important methods. Figure 3.2 displays the hierarchy of the Scala classes and traits of the `vertices` package[1], which encompasses all building blocks of the index graph. Traits are shown as square boxes and their use is depicted with dotted lines. Arrows with solid lines

[1]The full package name is `com.signalcollect.triplerush.vertices`

represent inheritance relationships. All classes, displayed as oval vertices, are declared either `abstract` or `final`: if the class is not extended any further, it is `final`, otherwise it is `abstract`.

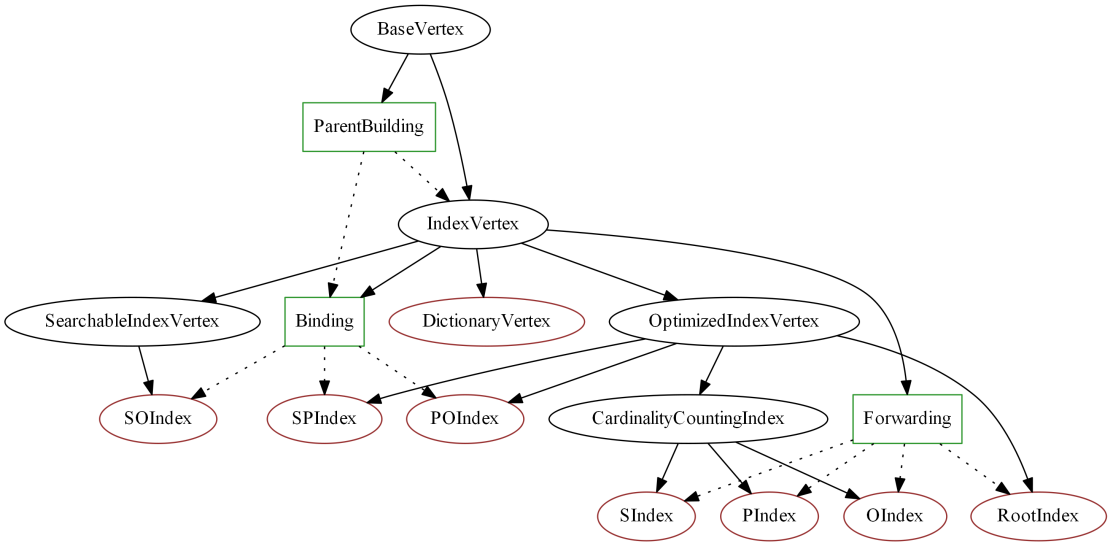

Figure 3.2: The hierarchy of the vertices package

The **BaseVertex** class extends the basic vertex class from Signal/Collect and abstracts some of the functionalities of Signal/Collect. Similarly, **IndexVertex** lays the groundwork for the extending classes, declaring an ID attribute which serves as unique identifier for all final classes and defining various abstract or empty methods, i.e. methods meant to be overridden at a later time where necessary. Such methods include: `processQuery`, overridden by the traits to handle an incoming query; `addChildDelta` such that a new child delta may be added to an index; and `cardinality` to be able to query indices for their cardinality (the number of child deltas). The method `deliverSignalWithoutSourceId` builds the heart of the communication – it handles incoming messages and decides what method to invoke based on the message type.

We see that SO-Index, SP-Index and PO-Index use the trait **Binding**, whereas the classes for S-index, P-index, O-index and the root index use the trait **Forwarding**. The first are commonly referred to as Binding indices, while indices of the latter group are so-called Forwarding indices.

The main task of the Forwarding indices is to forward incoming query particles (messages with information about the SPARQL query to process) to all of their children, e.g. the P-index `(*, continent, *)` forwards incoming query particles to its children, `(Switzerland, continent, *)` and `(Nepal, continent, *)`. The children are SP-indices and therefore part of the Binding indices, which, in turn, are responsible for binding values to the variables of the SPARQL query.

For instance, if the SP-index `(Nepal, continent, *)` receives a message wherein it must process the triple pattern `(?country <continent> ?cont)`, it binds the value

`Nepal` to `?country` and for each of its children, it binds the child delta to `?cont`. For every binding combination, it sends a message to the next destination, determined by the next triple pattern to process[2]. We discuss query evaluation in Section 3.4.

The traits `Binding` and `Forwarding` override the method `processQuery`, declared in `IndexVertex`. In `Forwarding`, the incoming query particle is divided into as many copies as there are children and a copy is sent to each child. `Forwarding` accesses the ID of the children through the method `nextRoutingAddress`, an abstract method in `Forwarding` which the Forwarding indices have to implement, i.e. the classes **SIndex**, **OIndex**, **PIndex** and **RootIndex**. This method allows each index type to define for which field the child deltas represent a value, e.g. a child delta stored in the root index designates a child of type P-index. In other words, the presence of a child delta `c` in `RootIndex` implies that a P-index `(*, c, *)` exists, which is a child of the root index.

On the other hand, the `Binding` trait is responsible for binding values to SPARQL variables. As in `Forwarding`, it creates as many copies as there are child deltas. For each child delta, adds bindings for the variables of a triple pattern. For example, given the triple pattern `(?country <memberOf> ?org)` and the PO-index `(*, memberOf, UnitedNations)` with child deltas `Switzerland` and `Nepal`, the variable `org` is bound to `UnitedNations`, and `country` will be set to `Switzerland` in one case and to `Nepal` in the other. Similar to `Forwarding`, it defines an abstract method `bindIndividualQuery` such that each Binding index may define what field the child delta stands for.

As mentioned at the end of Section 3.2, besides the task of binding variables, the **SOIndex** type is also used in place of SPO-indices to verify the existence of RDF triples. We see in Figure 3.2 that it is the only class inheriting from `SearchableIndexVertex`. This class handles the management of the child deltas and stores them in a structure which supports binary search for fast lookup, whereas in `OptimizedIndexVertex` – the class all other indices inherit from – the child deltas are managed in a splay tree.

3.4 Query Particles and Query Execution

We now take a detailed look at the *query particle* and how it is used to process queries. A query particle is a message that is passed among the indices of the index graph, in which data about the SPARQL query to answer is stored. It contains the triple patterns of the query and provides a means to save a binding for each variable, which are added by the Binding indices the query particle is routed to during query execution.

Entries are dictionary encoded as integers, and variable names in the query are substituted with negative integers. Therefore, we only require a collection of integers to represent and transmit all triple patterns of a query. The query particle is essentially an array of integers augmented with methods to ease the manipulation of the fields, such as to add a binding or to remove a triple pattern. Query particles are passed as simple `Int` arrays from index to index and implicitly converted to the Scala class `QueryParticle`[3]. The layout of the array is given in Table 3.1.

[2] Of course, in the case of `(Nepal, continent, *)`, there is only one child delta, `Asia`.
[3] The class `QueryParticle` is located in the package `com.signalcollect.triplerush`

0	1 - 2	3	4 to 4+b-1	t_1 to t_2
query ID	number of tickets	number of bindings b	bindings	triple patterns

Table 3.1: Indices of the query particle, where b = number of bindings, t_1 = 4+b, n = number of triple patterns, and t_2 = 4+b + n·3 - 1

A **query ID** is saved in field 0 of the array as to distinguish different queries which may be running at the same time over the index graph. The **number of tickets**, stored in indices 1 and 2, hold the number of times the query particle may be separated to pursue different paths. It is a number of type `Long`, saved as two `Int` fields in order to keep all query particle information in an `Int` array. For example, if the P-Index `(*, continent, *)` contains two children and receives a query particle, it will separate the query particle into two copies and the number of tickets will be divided among them. The number of tickets defaults to `Long.MAX_VALUE` and is reported by [Stutz et al., 2013] to be sufficient even for large datasets. Due to the asynchronous character of TripleRush, it is important to keep track of the number of tickets as TripleRush can only determine whether or not the execution has terminated by checking if all tickets have been sent back.

As previously mentioned, the query particle provides space to save bindings for the variables in the SPARQL query. The **number of bindings** `b` is saved in index 3 of the array. For instance, if a SPARQL query uses three variables, the number of bindings `b` is set to 3 and the three following fields – fields 4 through 6 (`4+b-1`) – are reserved to be able to store a binding for each variable. The fields for the bindings are organized by descending order of variable ID, i.e. the field at index 4 saves the binding for the variable encoded as -1, index 5 holds the value for -2, etc.

Finally, the **triple patterns** of the SPARQL query are saved at the end of the array, requiring three fields per triple pattern, either containing a dictionary encoded value or a negative integer to represent a variable. Since the number of bindings `b` may vary from query to query, the field in which the subject of the first triple pattern is stored, t_1, depends on `b` and is equal to `4+b`. The final field of the array, t_2, stores the object of the last triple pattern. It depends on the number of triple patterns `n` and can be computed as $t_2 = t_1 + n \cdot 3 - 1$.

Optimization. The triple patterns are ordered by an optimizer, based on estimations of how many results each element produces. The triple pattern expected to produce the least results is executed first and the triple pattern guessed to generate the most is processed last[4]. Triple patterns are added to the array of the query particle in reverse order; the first triple pattern to process is at the end of the array. Once the variables of the triple pattern have been bound, the triple pattern is removed from the array and the new triple pattern that is now at the end will be processed. This is repeated until the query particle does not contain any triple patterns anymore.

We now discuss how a query particle is processed through the index graph. Trivially, if a URI is present in a triple pattern which does not exist in the dictionary, we can

[4]This is the default optimizer, but TripleRush provides other, more elaborate approaches.

infer that it is not present in the dataset and therefore, we know that the query will not produce any results. In such cases, we immediately report that the query has no results and no query particle is created. This also avoids adding values to the dictionary which would have to be removed afterwards.

If a query passes the dictionary check, the execution starts with the addition of a *query vertex* to the index graph. This vertex serves as return point for all query particles of the SPARQL query. The query particle which was created is then sent to its first destination, determined by replacing the variables with wildcards in the first triple pattern to process. For instance, the triple pattern `(?country <continent> ?cont)` would be processed by the index `(*, continent, *)`.

Every index copies an incoming query particle as many times as it has children. The number of tickets is distributed equally among all copies and each query particle will continue its own path, with its individual combination of bindings. Forwarding indices simply forward a copy of the query particle to each of their children, while Binding indices add bindings to the query particle.

A Binding index uses its fields and its child deltas as values for bindings. For example, if the SP-index `(Switzerland, continent, *)` has to process the triple pattern `(?country <continent> ?cont)`, it will bind the variable in the subject field to its own subject, `Switzerland`, and the variable `cont` is bound to one of its child deltas.

During the binding process, not only is the field storing the binding of a variable set to a given value, but any occurrence of the variable in other triple patterns is replaced to the value as well. In other words, if the variable `cont` is bound to `Europe` and there exists another triple pattern where this variable occurs, e.g. `(?cont <population> ?pop)`, it is replaced to `(<Europe> <population> ?pop)`. If this is the next triple pattern to process, the query particle is sent to the SP-index `(Europe, population, *)`.

There is no guarantee that the next destination exists. In the example, if there is no index `(Europe, population, *)`, the tickets of the query particle are sent back to the query vertex to inform it that no result was found for the given number of tickets. However, if such an SP-index does exist, the variable `pop` would be bound to the child deltas of the index.

A query particle will either be sent to a destination that does not exist, whereupon the tickets are returned to the query vertex, or all triple patterns can be processed successfully, after which the query particle is sent back to the query vertex, reporting a result. Eventually, all tickets will be returned and the query vertex will be removed from the graph, as the query it is associated with has been executed.

3.5 A Query Example

We now take a detailed look at how a sample query is processed in TripleRush. Consider the RDF triples given in Table 2.1 and the following SPARQL query:

```
SELECT ?country ?pop
WHERE {
    ?country <continent> <Europe> .
    ?country <population> ?pop
}
```

We assume the dictionary encodings given in Table 3.2 and that the query ID is 2.

Value	ID	Value	ID
Switzerland	3	continent	5
Europe	7	Nepal	9
population	13	8183000	14

Table 3.2: Assumed dictionary IDs for the given values

The triple patterns can be processed in two different orders. As discussed in Section 3.4, the order of the triple patterns is usually determined by an optimizer and also depends on the dataset.

Starting with the continent triple pattern. If we assume that the triple pattern `(?country <continent> <Europe>)` has been selected to be handled first, the query particle which would be produced is given in Table 3.3. The query particle starts with the query ID, `2`, and then stores the number of tickets as a `Long` number by making use of two `Int` fields. Index 3 informs us that there are two variables to save bindings for – `country` and `pop` – and therefore, we know that the following two indices are reserved for the bindings. As the variables are not bound in the beginning, the fields are set to 0.

As of index 6, all remaining fields are triple patterns. In this case, they are `(-1 13 -2)`, followed by `(-1 5 7)`. We can decode these triple patterns to `(?country <population> ?pop)` and `(?country <continent> <Europe>)`, respectively, with the dictionary and the knowledge that -1 stands for `?country` and -2 for `?pop`.

0	1 - 2	3	4	5	6	7	8	9	10	11
2	`Long.MAX_VALUE`	2	0	0	-1	13	-2	-1	5	7

Table 3.3: Sample query particle

To start solving the query, TripleRush takes the last triple pattern of the query particle, replaces the variables with wildcards, and sends the query particle to the according index in the index graph – in this example, to `(*, continent, Europe)` (dictionary encoded as `(0, 5, 7)`). As the index only has one child, the query particle does not have to be separated into multiple copies. The variable `country` can be bound to the child delta, `Switzerland`, encoded as `3`. The last triple pattern is removed from the query particle.

After this first binding, the query particle has the form as shown in Table 3.4. We notice that field 4 now holds the new binding and that the last triple pattern has been removed. Additionally, the variable `-1` in the remaining triple pattern has been replaced to its binding, `3`.

0	1 - 2	3	4	5	6	7	8
2	`Long.MAX_VALUE`	2	3	0	3	13	-2

Table 3.4: Sample query particle after first binding

The query particle is now sent to its next destination, determined by the remaining triple pattern. As before, variables are replaced with a wildcard and so the query particle is sent to the index `(3, 13, 0)` (i.e. `(Switzerland, population, *)`). There, the variable `-2` can be bound as well and, as there are no more remaining triple patterns, the query particle is sent back to the query vertex to report the result.

Starting with the population triple pattern. Alternatively, we assume that the triple pattern `(?country <population> ?pop)` has been given precedence. We determine the index to start at by replacing the variables in the triple pattern, `(-1 13 -2)`, with wildcards, yielding the index `(0, 13, 0)`, i.e. `(*, population, *)`. For the sample triples in Table 2.1, `(*, population, *)` has two children, namely the SP-indices `(Switzerland, population, *)` and `(Nepal, population, *)`. Therefore, two copies of the query particle are produced, each carrying half of the tickets, and they are routed to their next destination. Note that `(*, population, *)` is a Forwarding index – it neither adds bindings, nor does it remove triple patterns from the query particle; it merely forwards incoming query particles to its children indices.

One query particle is forwarded to the index `(Switzerland, population, *)` while the other goes to `(Nepal, population, *)`. Being Binding indices, these will remove the last triple pattern of their query particle and bind the two variables present in it. Binding indices use their own values where possible, i.e. `Switzerland` and `Nepal` are bound to `?country` in the respective query particle. Both indices only have one child delta, namely the population of the country, so the query particle does not have to be separated and `?pop` is set to the according population number in either query particle.

The query particle is shown in its unbound form and in its bound form for `Switzerland` in Table 3.5. We see that the variable `-1`, which was present in the remaining triple pattern, has been replaced with the binding, i.e. the remaining triple pattern has been changed from `(-1 5 7)` to `(3 5 7)` for `?country = Switzerland`. The query particle at `(Nepal, population, *)` contains one triple pattern of the form `(9 5 7)` after the binding process.

	0	1 - 2	3	4	5	6	7	8	9	10	11
Beginning:	2	`Long.MAX_VALUE`	2	0	0	-1	5	7	-1	13	-2
Bound variables:	2	`Long.MAX_VALUE/2`	2	3	14	3	5	7			

Table 3.5: Query particle for alternative route

Although the remaining triple patterns in both query particles do not contain any variables anymore, they must be checked nonetheless and cannot be simply discarded. The query particles are forwarded to their next destination, determined by the last triple pattern. As it does not contain any variables anymore, it is routed to the corresponding SO-index by replacing the predicate with a wildcard. We see why this step is necessary in the query particle where `?country` is assigned to `Nepal`. The remaining triple pattern is `(9 5 7)` and so its destination is `(9, 0, 7)` – i.e. `(Nepal, *, Europe)`. However, as no such vertex exists, the query particle is halted and the tickets are sent back to the query vertex without any result.

Even if the index `(Nepal, *, Europe)` existed (for example, because there exists a triple `(Nepal, likes, Europe)` in the stored data), the query particle would still be halted, as the SO-index would check that it has a child entry for `(Nepal, continent, Europe)`. Since this is not the case, the tickets are sent back to the query vertex with no results; otherwise, as in the query particle where `?country` is bound to `Switzerland`, the query particle is sent back to the query vertex to report the successful binding.

The paths the query particles go through in either possibility are shown in Figure 3.3. If we start with the triple pattern that limits the results to European countries, we see that we never have to generate any copies of the query particle path during execution, as both indices the query particle travels through only have one child. On the other hand, if we start by selecting the population for all countries, the query particle is divided into two copies in the index `(*, population, *)` as it has two children, enabling each query particle to pursue its route for its individual binding.

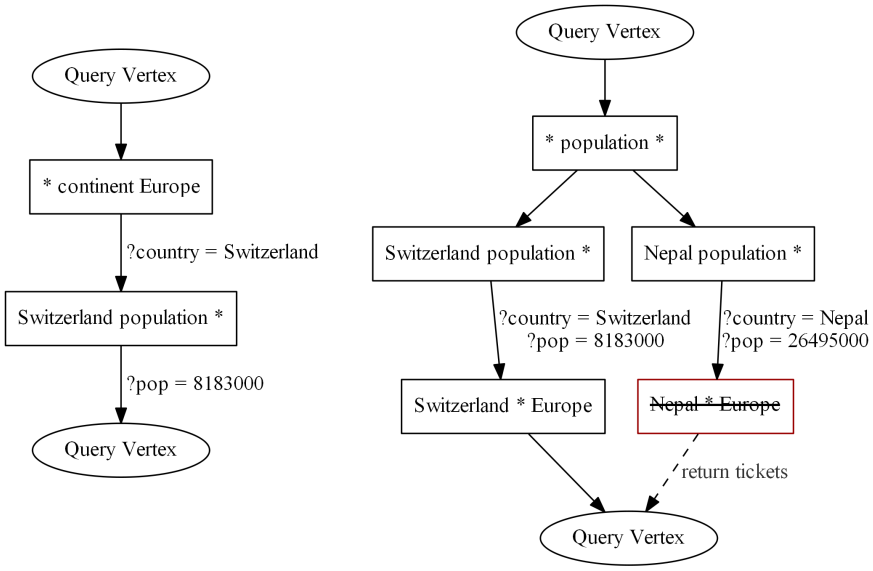

Figure 3.3: Query particle paths for both orders of execution.
Left: Query particle path when selecting European countries first.
Right: Query particle paths when starting with the population triple pattern.

4

Implementation of Filters

In this chapter, we describe the implementation of SPARQL filter support in TripleRush[1], from parsing the SPARQL query to extract filter information to the subsequent evaluation of the filters. Prior to this thesis, SPARQL filters were not recognized by TripleRush and attempting to evaluate queries with filter clauses would result in an error.

An additional parser has been added to TripleRush to parse the information from the filter clauses. It is discussed in Section 4.1. The index graph of TripleRush has been expanded to include a new vertex, namely the dictionary vertex, which is discussed in Section 4.2. Filters are represented as trees, designed with modularity in mind. Additional elements can be added easily and there is no need to need to know the actual layout of the trees to evaluate the filters. They are presented in Section 4.3.

4.1 SPARQL Filter Parsing

The first step to processing a SPARQL query is to parse it, such that the conditions and constraints of the query can be understood. TripleRush comes with its own SPARQL parser for this purpose – a custom parser extended from the `Parsers` family of traits in Scala. The parser has been extended to recognize filter clauses among the triple patterns. However, it does not parse the filter body (the text in the `FILTER` clause) any further due to the extensive grammar of SPARQL filters.

After the SPARQL query has been generally parsed, the extracted filter bodies are sent to an additional parser, the filter parser, which checks the filter bodies for syntactical validity and transforms them into a representation which TripleRush can evaluate. The filters are transformed into *filter trees*, whose structure corresponds to a simplified version of the grammar provided in the SPARQL specification [Prud'hommeaux and Seaborne, 2008].

Terms in the original grammar which merely refer to another term have been skipped in the implementation of the filter parser. For instance, consider the following two grammar rules in the SPARQL specification:

In the filter parser, these rules have been merged as one – `ConditionalAndExpression` is defined as `RelationalExpression ( '&&' RelationalExpression )*`. This helps keep the parser and the resulting trees free of unnecessary elements.

[1] *https://github.com/jacqueslk/triplerush-filter*

[48]	ConditionalAndExpression	::=	ValueLogical ('&&' ValueLogical)*
[49]	ValueLogical	::=	RelationalExpression

The constraints supplied in the filters may contain multiple arithmetic and boolean operators, such as `FILTER(?A*3 = ?B+2 && ?B > 0)`. Trees are therefore a suitable structure to represent the complex combinations of the many elements SPARQL filters may consist of. Variables are identified by the same numbers as in the query particle (see Section 3.4), i.e. if a variable such as `?pop` is encoded as -2, the variable is identified by -2 in the filter tree as well.

Figure 4.1 shows the representation of the filter `FILTER(?pop > 10000000)`, given that `?pop` is encoded as -2.

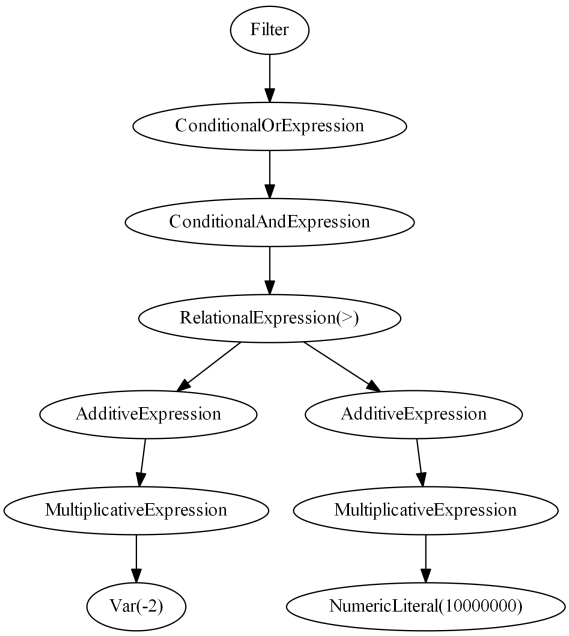

Figure 4.1: Filter tree for `?pop > 10000000`

The different elements of the filter trees are discussed at length in Section 4.3. Once the filter parser has constructed the filter trees, they are returned to the main SPARQL parsing class and saved with the remaining data which were parsed from the query.

After the SPARQL query has been parsed successfully, we construct a query particle as presented in Section 3.4. Due to the complexity of SPARQL filters, we do not save any filter information to the query particle as it consists only of an `Int` array. Instead, we send the filter trees of the query to the so-called dictionary vertex, which allows us to check the filters for every new binding during the execution of the query.

4.2 The Dictionary Vertex

The *dictionary vertex* is an index in the index graph which has access to the dictionary of TripleRush. During query execution, query particles are sent to the dictionary vertex each time one or more variables have been bound. The dictionary vertex checks all relevant filters for the new bindings in the query particle. If any filter fails, the dictionary vertex halts the query particle and sends the tickets back to the query vertex. Otherwise, the query particle is sent to its next destination.

4.2.1 Communication

Once a SPARQL query has been parsed, a query vertex is created and added to the index graph. Before sending the initial query particle to its first destination, the query vertex sends the list of filters in the query to the dictionary vertex, along with the query ID. The dictionary vertex saves the list of filters with the query ID as key for future referral. At the end of the query execution, before the query vertex is removed, it sends a message to the dictionary vertex to signal that the filters can be removed.

During query execution, multiple copies of the query particle may be created whereby each follows an individual path, i.e. each copy holds a unique set of bindings for the variables in the query (cf. Section 3.4). To expand TripleRush to support filters, we need to guarantee not only that all triple patterns are satisfied, but that the bindings also fulfill the conditions of the filters.

To achieve this, we send query particles to the dictionary vertex every time a new binding is saved. Since bindings are only added in indices with the `Binding` trait, we only need to change the routing system in the `Binding` trait to be aware of the dictionary vertex. Essentially, there is only one situation in which it is necessary to send the query particle to the dictionary vertex: when new bindings have been added.

However, we differentiate between three cases: (a) a new binding is saved to the query particle of a system query, (b) a new binding is saved to a query which is now completely solved and planned to return to the query vertex, and (c) a new binding is stored to a query particle which will be sent to another index for further processing.

In case (a), system queries, we can immediately send the query particle to its next destination without any interference. System queries are launched by TripleRush itself to gather statistics about the RDF data and the index graph (e.g. to get the cardinality of an index) and never make use of SPARQL filters. System queries are easily identified as they are the only ones to use negative query IDs.

In both remaining cases, (b) and (c), we must contact the dictionary vertex at some point to evaluate the filters. We append the IDs of all newly bound variables to the query particle array and also add the next destination of the query particle, i.e. the ID of the vertex the query particle should be sent to after passing through the dictionary vertex. For case (b), the query particle the next destination is the query vertex, while for (c), it is another index of the index graph.

For case (b), we directly send this modified query particle to the dictionary vertex. The dictionary vertex will remove the additional information from the query particle before

processing the filters and finally transmitting the query particle to the destination it was provided, if all filters passed.

The final case, (c), covers the addition of bindings to a query particle which is not fully solved yet, i.e. it will be sent to another index, which will process another triple pattern of the query particle. In this case, we wrap the modified query particle into a so-called *FilterPending* object and send it to the next destination, as determined by the query particle.

At the arrival of a FilterPending message, the index simply sends the query particle it contains to the dictionary vertex, akin to what is immediately done in case (b). Again, the dictionary vertex will extract and remove the additional information from the query particle, evaluate the relevant filters, and send the query particle to the destination it was given if the filters pass.

Handling these three cases separately offer us considerable advantages. In the case of (a), it is evident that the query will never have any filters, so we save overhead by immediately sending the query to its next destination. In the case of (b), we know that the query vertex exists and we avoid modifying any code in the query vertex classes by handling all filter matters beforehand.

Contrary to the latter case, for (c), we send a FilterPending message to the next index the query particle should continue to, rather than sending the query particle directly to the dictionary vertex for evaluation. The reason behind this seemingly inefficient design is the simple fact that there is no guarantee that a query particle's next destination actually exists. We recall from Section 3.4 that the next destination of a query particle is determined by the triple pattern which will be processed subsequently. However, such a triple patterns may contain bound variables, which do not necessarily satisfy the next triple pattern. In such situations, it is possible that the query particle should be sent to an index which simply does not exist in the index graph[2]. In those cases, the tickets of the query particle are sent back to the query vertex to signal that no results have been found for that portion of tickets. By sending a FilterPending message to the destination index, we avoid checking filters for a query particle which will be forwarded to a nonexistent index.

4.2.2 Filter Evaluation

As soon as the dictionary vertex receives a list of filters for a new query, it checks whether any filters do not contain any variables, i.e. if they can be evaluated directly without any binding information. The dictionary vertex processes all such filters immediately and removes them from the collection. If a filter with no variables cannot ever be true, e.g. `FILTER(1 > 5)`, a *global false filter* is placed at the start of the filter list. The global false filter is a simple filter which always evaluates to `false`; it signals to the dictionary vertex that it should immediately halt any query particles of the query and simply send back the tickets to the query vertex.

[2]For example, in Section 3.5, we explore how a sample query is solved in TripleRush and bind `?country` to `Nepal` in one query particle. The next triple pattern, `(?country continent Europe)`, dictates that the query particle should be sent to the index `(Nepal, *, Europe)`, but no such index exists.

Query particles are sent to the dictionary vertex every time a new variable is bound. Therefore, it is possible that the same query particle is sent to the dictionary vertex on multiple occasions. To avoid evaluating any filters for the same query particle more than once, we also send a list of the variable IDs which have been newly bound to the dictionary vertex. Thanks to this information, the dictionary vertex is able to select the filters which require one of the newly bound variables and to evaluate only these.

The dictionary vertex can query each filter for its *variable set*, i.e. for the list of variables required to evaluate the filter. The dictionary vertex determines whether or not a filter should be evaluated by checking that the variable set contains at least one newly bound variable, and by ensuring that the filter does not require any variables which have not yet been bound. The remaining filters were either checked previously, or they cannot be checked yet as another required variable still has to be bound.

If a filter is deemed relevant, the dictionary vertex must retrieve the actual values of the bindings and pass them to the filter. The bindings are saved as dictionary IDs in the query particle. All other indices of the index graph are only aware of the dictionary IDs and do not know the actual values. In contrast, the dictionary vertex has access to the dictionary of TripleRush, hence its name. Therefore, it can retrieve the actual values of the required variables to process the filter.

Filters are evaluated by passing the real values of the variables to a method in the `Filter` object. The filter responds with a boolean value, informing the dictionary vertex whether or not the filter passes with the given values. If all checked filters pass, the dictionary vertex sends the query particle to its next destination. If any filter fails, the dictionary vertex sends the number of tickets back to the query vertex, indicating that no result was found for the returned tickets.

Since filters with no variables are immediately evaluated and query particles are sent to the dictionary vertex every time a new binding has occurred, we can guarantee that all filters in the list will have been evaluated exactly once for each query particle before it is sent as a result to the query vertex.

4.3 Filter Trees

We now take a more technical look at the filter trees. As discussed in Section 4.1, trees are a suitable structure to save filter information as they allow us to represent even complex forms of constraints. We show the filter tree for `FILTER((?A+10)*5 > 200 && (?A || 10 < ?B))` in Figure 4.2 as an example. We see that, especially when comparing it to the simpler tree shown in Figure 4.1, the tree is simply expanded as the constraints become more complex.

The root of every filter tree is a `Filter` object. It contains the necessary methods to use the SPARQL filter the tree represents, without having to consider the underlying structure. Leaf nodes represent variables and literals. All remaining nodes – the internal nodes, aside from the root – may have one or more children and represent a type of relationship between the child nodes. For example, a comparison such as `?pop > 10000000` is modeled as a node of type `RelationalExpression`; it stores two children, `?pop` and

`10000000`, and the type of the relationship (e.g. comparison with `>`). Trivially, if no such comparison is present in the SPARQL filter, the node only has one child, and the value the node evaluates to is equal to that of its only child.

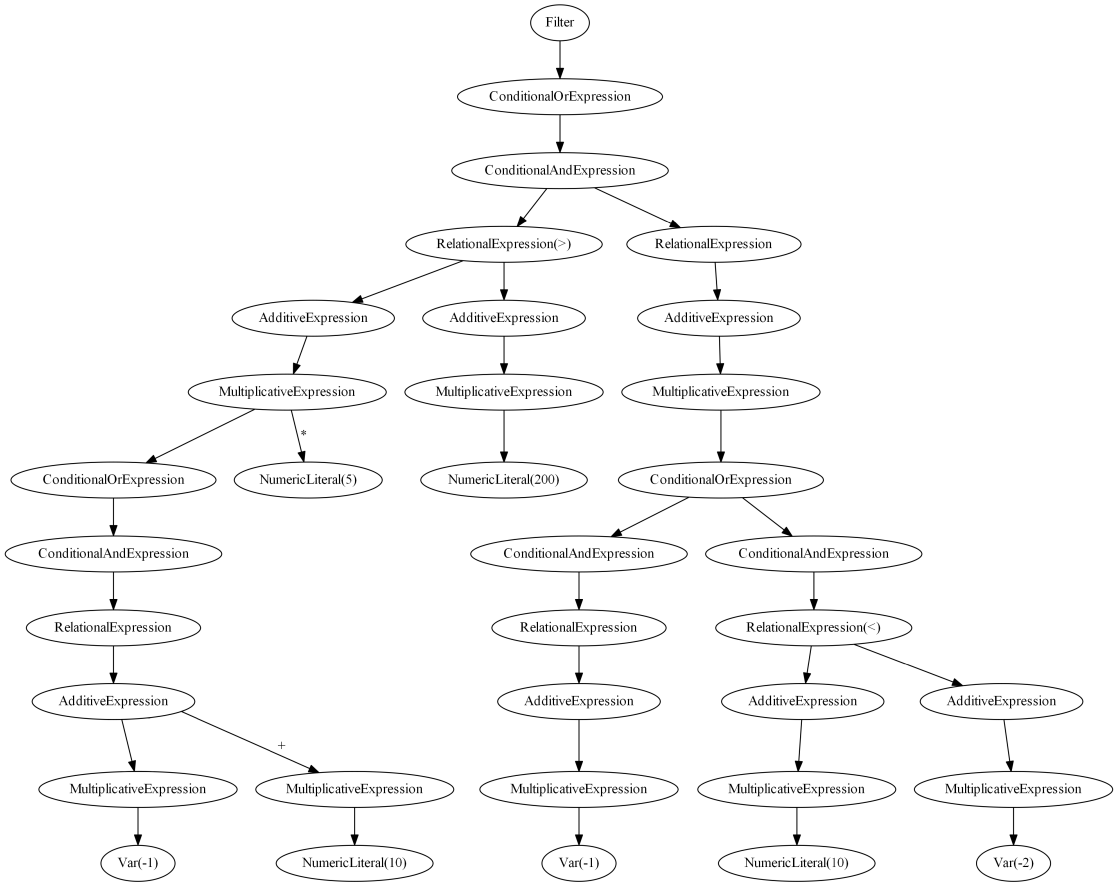

Figure 4.2: Filter tree for `(?A+10)*5 > 200 && (?A || 10 < ?B)`

4.3.1 Filter Elements

Each element of the filter tree is represented by a Scala class; the names are taken from the grammar published in the SPARQL specification [Prud'hommeaux and Seaborne, 2008]. It is thus easy to understand what type of information an element contains by referring to the SPARQL specification.

Every filter is represented by an object of the class `Filter`. Following the principles of encapsulation, it is sufficient to know the methods in `Filter` to use such an object during query evaluation, e.g. in the way the dictionary vertex does. The filter class contains two methods: `getVariableSet()` and `passes()`. The method `getVariableSet()` returns

a Scala `Set()`[3] of all variables present in the filter. With this method, it is possible to determine whether the necessary variables have been bound to evaluate the filter. The method `passes()` takes a list of bindings for the variables it requires as parameter and returns whether or not the filter passes.

All classes for internal nodes of the filter tree contain the methods `getVariableSet()` and `getValue()`. A given element's `getVariableSet()` method typically calls the `getVariableSet()` method of its children. For instance, given the filter tree in Figure 4.1, calling `getVariableSet()` of the `RelationalExpression` object invokes the method `getVariableSet()` of both `AdditiveExpression` objects and returns the union of the sets. Thus, it would receive `Set(-2)` from the child on the left-hand side and an empty set from the right one. The result is therefore `Set(-2)`.

The **Var** class represents a variable identified by the variable ID, a unique negative integer assigned to each variable during the parsing process. **NumericLiteral**, in turn, represents a concrete value provided in the filter. These two classes are part of the *primary expressions* as given in the official SPARQL grammar in [Prud'hommeaux and Seaborne, 2008]. We realize this rule by making the classes inherit from the trait **PrimaryExpression**.

A **MultiplicativeExpression** object saves one or more primary expressions and represents consecutive multiplications and divisions among such items, e.g. `?A*5/?B`. One level above in the tree hierarchy, the **AdditiveExpression** class represents one or more multiplicative expressions connected by consecutive additions or subtractions, such as `5+?A` or `?A*3 - ?B/2`, where `?A*3` and `?B/2` are multiplicative expressions.

Each **RelationalExpression** object consists of one or two additive expressions. Trivially, if it only contains one additive expression, it simply forwards the value its only child evaluates to. If it has two children, it represents the boolean result of one of the binary operators `=`, `!=`, `<`, `>`, `<=`, or `>=`. A relational expression can thus represent a comparison such as `?pop > 1000`, where `?pop` and `1000` are trivial additive expressions.

Finally, **ConditionalAndExpression** represents one or more relational expressions which are concatenated with the Boolean AND operator (&&). In other words, if such an object stores multiple elements, it only evaluates to true if all children evaluate to true. Similarly, a **ConditionalOrExpression** object stores one or more `ConditionalAndExpression` objects which are combined with the Boolean OR operator, ||. When it contains several conditional-and expressions as children, it returns true if at least one element evaluates to true, too.

[3] A set is a collection in which each element may only appear once.

4.3.2 Effective Boolean Value and Combination of Elements

The inner nodes (besides the root) represent a specific type of relationship between their children. If no such relationship exists – e.g. if there is no && operator in the filter – the node has only one child. Furthermore, bracketed expressions such as `(?A+5)` in `(?A+5)*3` represent an additional challenge. Consider the following rules from the SPARQL grammar[4] [Prud'hommeaux and Seaborne, 2008]:

[55]	PrimaryExpression	::=	BracketedExpression | NumericLiteral | Var
[56]	BracketedExpression	::=	'(' Expression ')'
[46]	Expression	::=	ConditionalOrExpression

A bracketed expression can act as a primary expression, along with variables and literals. We note that a bracketed expression is simply a ConditionalOrExpression within brackets and therefore, we simply state that a `ConditionalOrExpression` is also of type `PrimaryExpression` in the filter trees. The SPARQL parser in TripleRush still checks for brackets but simply discards them and creates a `ConditionalOrExpression` object. In this manner, we guarantee that we still follow the rules of the SPARQL grammar but we can avoid trivial `BracketedExpression` or `Expression` nodes, which would only evaluate to the value of the only child they could have.

A filter must result in a boolean value, i.e. the result of the `Filter` object's `passes()` method must always be a boolean. It is, however, possible to have a filter body of numeric type, such as `FILTER(?A+5)` or simply `FILTER(-3)`. In cases where a non-boolean value must act as a boolean, the so-called *effective boolean value* is used [Prud'hommeaux and Seaborne, 2008]. For instance, numeric values evaluate to false only if the value is NaN or equal to 0, and to true otherwise.

The filter tree of a simple filter like `FILTER(-3)` consists of a root `Filter` node and the remaining internal nodes are trivial nodes of the following types, given in order: ConditionalAndExpression, ConditionalOrExpression, RelationalExpression, Additive-Expression and MultiplicativeExpression. The last node contains one child – the tree's only leaf node – of type NumericLiteral, which represents the number -3 given in the filter.

Calling the root node's `passes()` method will make it invoke its child's `getValue()` method. The child is of type ConditionalOrExpression and retuns a boolean value if it has multiple children: `true` if at least one child evaluates to true, or `false` otherwise. In our case, the ConditionalOrExpression node only has one child, in which case it simply "forwards" the value of its only child, regardless of the type. In this example, ConditionalOrExpression evaluates to -3 and passes this to the `Filter` object. The `Filter`'s `passes()` method is required to return a boolean, and therefore, the effective boolean value of the result is returned, i.e. the filter passes.

Various nodes may require their children to evaluate to boolean values if they have multiple children. All such cases are given in Table 4.1. Keeping the original value as long as possible – rather than forcing all ConditionalOrExpression nodes to evaluate to

[4]Some members of the rule for `PrimaryExpression` have been removed in the listing for brevity.

a boolean value, for instance – allows us to support bracketed expressions for arithmetic operations. For example, the left-most `ConditionalOrExpression` node in Figure 4.2 represents the bracketed expression `(?A+10)` in the expression `(?A+10)*5 > 200`.

Class	Condition	Represents
Filter	Always	Result of the filter
ConditionalOrExpression	Multiple children	Boolean operator ||
ConditionalAndExpression	Multiple children	Boolean operator &&
RelationalExpression	Two children	Boolean comparator (e.g. !=)

Table 4.1: Cases when a node uses the effective boolean value of its children.

Finally, we recognize that not all SPARQL filters which conform to the SPARQL grammar can be evaluated. For instance, a filter such as `FILTER(?A*(?B>10))` does not break any grammar rules and a filter tree can be created. However, the bracketed expression `?B>10` returns a boolean value and cannot be multiplied to `?A` as it is only possible to multiply two numeric values.

According to the SPARQL specification [Prud'hommeaux and Seaborne, 2008], such expressions should return the value *error*, an additional value to the two boolean ones, and the entire filter may evaluate to the error value. In TripleRush, every time an error occurs, i.e. in any case where the SPARQL specification tells us to return the error value, it is reported to the user with specific information about the error, and we use boolean `false` instead.

4.3.3 A Filter Evaluation Example

We now make an example of how a filter tree is used to determine whether or not a given set of bindings satisfy the SPARQL filter the tree represents.

Given the filter `FILTER(?pop > 10000000)`, whose filter tree is shown in Figure 4.1, we assume that `?pop`, encoded as -2, is bound to the value 8,000,000 in a query particle. To invoke the `Filter` object's `passes()` method, we must pass the binding for all variables present in the filter. We query the filter for the variables it contains by calling its `getVariableSet()` method. In this example, the filter returns `Set(-2)` – a set with the ID of the only variable it contains.

Bindings are stored as dictionary encoded IDs in the query particle. The dictionary vertex retrieves the ID of the binding for `?pop` from the query particle, with which it can look up the actual value of the binding from the dictionary. It saves the bindings in a Map and passes all of the retrieved bindings to the filter's `passes()` method. In our example, the Map contains one entry with key `-2` and 8,000,000 as value.

The filter, in turn, invokes the `getValue()` method of its child, i.e. of the ConditionalOrExpression node, and passes the list of bindings along. ConditionalOrExpression only has one child and therefore simply evaluates to what its child evaluates to. Thus, it calls `getValue()` of its only child, the ConditionalAndExpression node, which, in turn, invokes the `getValue()` method of the RelationalExpression node.

The RelationalExpression node has two children and has to check whether the left child is greater than the one on the right-hand side. First, it must know the values of both children, so it calls the `getValue()` method of its AdditiveExpression children. The left AdditiveExpression node simply refers to the value of its child, which again refers to its child, the Var node. Each time a parent calls its child's `getValue()` method, it supplies the list of bindings, and therefore, the Var node can look up its binding and return it as result. The bound value of the Var, 8,000,000, wanders up from Var to MultiplicativeExpression and from there on back to AdditiveExpression, which reports it as its value to RelationalExpression. Similarly, the right AdditiveExpression node reports 10,000,000 as its value, which was returned from the NumericLiteral node.

Knowing the value of its children, the RelationalExpression node can now evaluate the expression and checks whether 8,000,000 is greater than 10,000,000. As this is not the case, it evaluates to false and the boolean value wanders all the way up to the root, the `Filter` object. As the value is a boolean, the value can be returned as is and the invoker of the `passes()` method is informed that the filter failed.

4.4 Conclusions

In this Chapter, we have presented how filters are transformed and processed. The filter tree structure allows us to represent and evaluate even complex forms of constraint.

The filter trees have been built in a modular fashion, such that it is not necessary to know the structure of the tree to evaluate the filter. Additionally, all elements of the trees only know about their direct children. As such, it is easy to introduce additional elements to the tree or to change the implementation of certain nodes.

Finally, we evaluate all filters as soon as possible to directly discard any unsatisfactory query particles. Not only are filters processed as soon as possible, but we also guarantee that we do not process any filters multiple times for the same query particle.

5

Evaluation

In this chapter, we discuss the performance of TripleRush, especially in regards to queries with SPARQL filters.

All experiments were conducted on one machine with 128 GB RAM and two E5-2680 v2 at 2.80 GHz processors, with 10 cores per processor. We ran the evaluation with version 1.8.0_25 of the Java Runtime and allocated 30 GB to the Java virtual machine (JVM) in every run.

The data for the evaluation stems from the *Berlin SPARQL Benchmark* (BSBM) [Bizer and Schultz, 2009]. Datasets and use case queries were created with the standard generator from BSBM. The evaluation was performed on five datasets of varying size, namely on a total of 100, 666, 2785, 28850 and 70812 RDF triples, respectively. Prior to each evaluation, 20 queries were run as warm-up queries, i.e. they were executed in the beginning but no statistics were gathered from them.

Ten subqueries were executed for each query category in BSBM. SPARQL elements which are currently not implemented in TripleRush have been removed from the queries, notably the `OPTIONAL` clauses, string and date filters, the `OFFSET` keyword, and type casts in the `ORDER BY` clause[1]. Queries 9 and 12 have been excluded from the evaluation, as TripleRush lacks support for the `DESCRIBE` and `CONSTRUCT` query forms.

We start measuring the execution time of a subquery when the SPARQL query is passed to the parser, and we consider the execution to have terminated once the returned results have been iterated through. The results are dictionary encoded; looking up the actual values from the dictionary is not included in the execution time. We compute the average of all subqueries for each query category individually and report the average of these figures as the average execution time for each dataset.

To contrast the time it takes to evaluate the SPARQL filters, we also evaluated the same queries without any filters. The comparison is shown in Figure 5.1.

We notice a drastic increase in execution time for the variant with the SPARQL filters as the datasets grow. Especially in query 5, there is a drastic difference between the execution time of the filter variant and the one in which the SPARQL filters were omitted. For instance, in the dataset with 28850 triples, executing the queries with filters took an average of 883 ms (best time: 333 ms), whereas omitting the filters from the query required a mere 30 ms on average (best time: 17 ms). The average execution times over query 5 is given in Figure 5.2.

[1] In query 10, `ORDER BY xsd:double(str(?price))` has been replaced with `ORDER BY ?price`

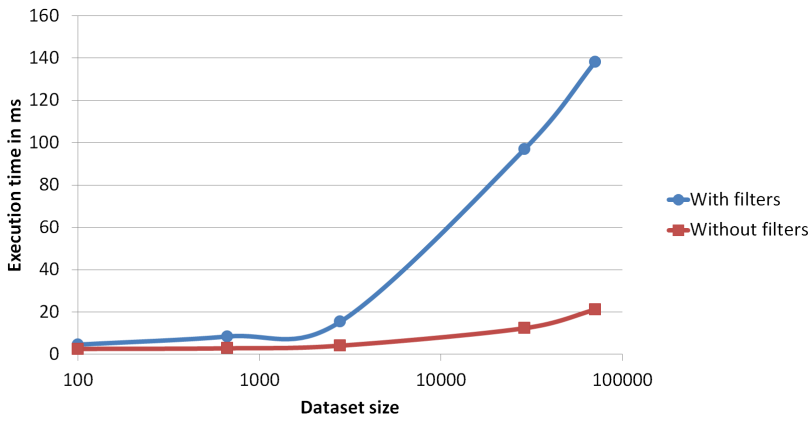

Figure 5.1: Average execution time over BSBM datasets

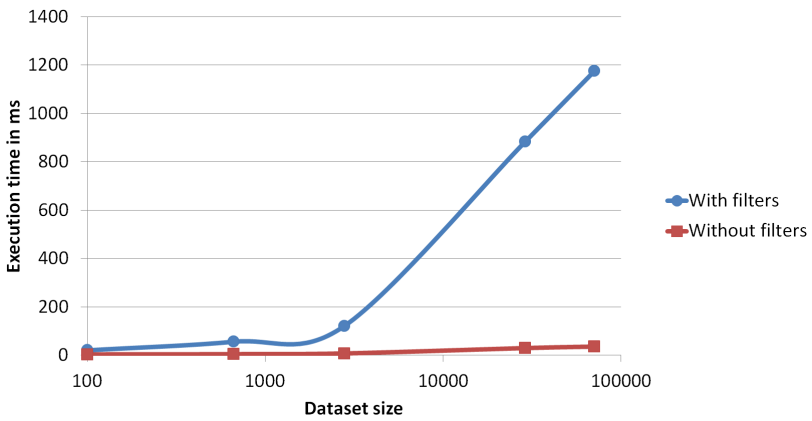

Figure 5.2: Average execution time of query 5

An increase in execution time is to be expected for SPARQL queries which contain filters, as they represent more criteria which TripleRush must take into account. In order to evaluate the filters for a query particle, the dictionary vertex must decode the dictionary IDs and pass the real values of all variables to the filter tree. The filter trees resulting from complex SPARQL filters consist of numerous nodes, which all take part in the evaluation of the filter.

We note that query 5 contains the following complex filters[2].

```
FILTER (?sim1 < (?orig1 + 120) && ?sim1 > (?orig1 - 120))
FILTER (?sim2 < (?orig2 + 170) && ?sim2 > (?orig2 - 170))
```

In contrast, the subqueries of query 1 only contain a simple arithmetic filter, such as `FILTER(?value1 > 214)`, whereby the number the variable is compared to differs in each subquery. With such queries, we do not observe large differences between the execution of queries with and without filters. The average execution times are given in Figure 5.3.

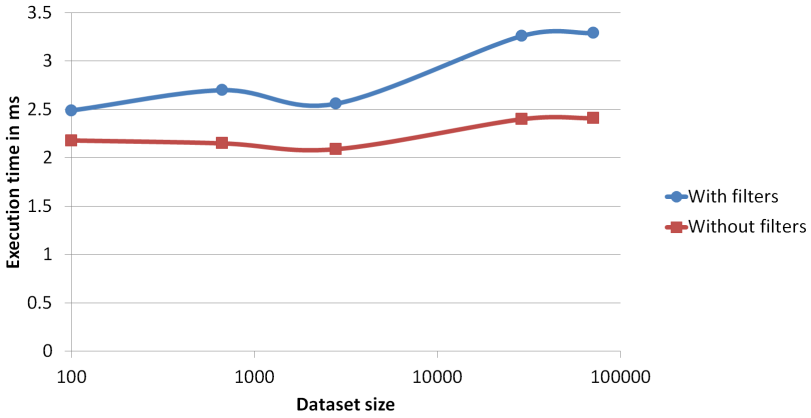

Figure 5.3: Average execution time of query 1

Aside from the apparent difference in filter complexity, we notice other differences between query 1 and query 5. The latter is specified to never return more than five results with `LIMIT 5`, whereas query 1 includes the constraint `LIMIT 10`. However, query 5 always returns five results, even when executed over the smallest dataset with SPARQL filters. Query 1, on the other hand, usually returns less than the allowed maximum of ten results. For instance, ten results were only reported once when we executed ten subqueries over the dataset of 28850 triples, even when the SPARQL filters were discarded.

[2]Note: The variable names have been shortened for layout purposes.

Therefore, we can assume that there are fewer query particles in query 1 which the dictionary vertex has to process. In query 5, there are many copies of the query particle, and the dictionary vertex acts as a central bottleneck as all newly bound query particles must traverse through it.

We can conclude that the ideal time to evaluate filters depends on the type of the query. Especially if there is a low `LIMIT` to a query with many results and lenient filters, it is evident that processing the filters afterwards is more efficient. On the other hand, if filters are strict and prevent many copies of the query particle to spawn which will not produce any end results, the current design for filter evaluation, i.e. evaluating after any new binding, is the most efficient.

6

Conclusion and Outlook

The result of the work during this thesis is the implementation of SPARQL filter support[1]. Particularly, TripleRush has been expanded to be able to parse and to transform SPARQL filters into a flexible, machine-processable format and to incorporate the process of checking filters into the overall execution of SPARQL queries. The filter trees have been designed in a modular fashion, allowing developers to include additional filter elements with ease and few modifications.

The parser and the structure of the filter trees follow the SPARQL specification closely, aside from the discussed "element shortcuts" where multiple rules are summarized into one. Furthermore, the elements of the filter attempt to follow the specification as closely as possible during evaluation, e.g. by applying the effective boolean value as late as possible to retain the original value a branch evaluates to.

Moreover, the additions to the TripleRush codebase are well-documented and test cases are provided to verify that the parser and the overall filter evaluation behaves properly and that the new features are well integrated into TripleRush.

Additionally, this thesis sheds some light on the general mechanics of TripleRush and its layout, especially on the novel index graph it employs and may hopefully serve as a useful reference for newcomers to TripleRush.

In conclusion, we note some areas in the filter processing mechanism of TripleRush that will hopefully see additions or improvements in the future.

Inefficient aspects. (a) During the binding process, the list of newly bound variables is computed for each child delta, while it would suffice to determine the newly bound variables beforehand. (b) TripleRush does not dedicate a field in the query particle for the binding of variables which are not declared in `SELECT`. Filters, on the other hand, may require variables absent from the `SELECT` clause. As a quick fix, the smallest variable ID present in a filter is used to determine the number of bindings to save in the query particle. For example, a query may select two variables, encoded as -1 and -2, and a filter may require a variable encoded as -7. In such a case, the query particle is expanded to maintain a field for the binding of all variables from -1 to -7, rather than only for -1, -2 and -7. (c) At the arrival of filters, the dictionary vertex checks whether any do not depend on any variables. This is achieved by calling the variable set of each filter and checking whether or not it is empty. However, it would be sufficient to stop the execution

[1] *https://github.com/jacqueslk/triplerush-filter*

as soon as the first variable has been found. (d) Queries which do not contain any filters do not have to be sent through the dictionary vertex. A possible way of avoiding routes to the dictionary vertex would be to let the query IDs of filterless queries be above or below a given number. For example, all queries with filters could have IDs above 500.

Type system. Presently, the dictionary saves all values as strings and it is therefore impossible to distinguish true strings from other datatypes. For instance, we cannot tell whether the original data was a string of digits (e.g. "15") or an actual number (e.g. the number 15). Moreover, it is necessary to reconstruct datatypes such as `xsd:dateTime` from their string representation.

Currently, we assume that all bindings provided to the filter tree are of numeric type. All numbers are treated as doubles in the filter trees to easily perform arithmetic operations, to avoid checking for cases such as integer overflow or division among two integers, where we would have to switch to doubles.

Further filter features. Certain SPARQL filter types are not yet supported in TripleRush, notably built-in functions and external functions. The issues with the type system render the implementation of functions like `isIRI` problematic. Furthermore, TripleRush's lack of support for `OPTIONAL` clauses currently renders the use of the function `bound` meaningless.

General optimization. Optimization for queries with filters might not be the same as for queries without filters. The estimations of the optimizer may need changes for better results, and it may be desirable to store additional meta information for more accurate estimates.

Filter optimization. While we evaluate filters which do not contain any variables once and then discard them if they pass (or we block query processing if any fails), potential for optimization lies in merging nodes of the filter trees which do not depend on any bindings, such as optimizing `FILTER(3+5 > ?A*3*(2+2))` to `FILTER(8 > ?A*12)`, or recognizing that `FILTER(?A>5 && 3>4)` will never evaluate to true. Similarly, it may be desirable to recognize when filters are tautologies, such as `FILTER(?A = ?A)`, or contradictions like `FILTER(?B-?B > 0)`.

As discussed in Chapter 5, it may be interesting to evaluate the filters as a post-processing step in certain situations, e.g. when a query has many triple patterns which numerous copies of the query particle would not satisfy.

Another potential for speed may lie in reordering multiple filters in an optimal order, or even the children of ConditionalAndExpression and ConditionalOrExpression nodes. This, of course, is a complex task and requires meta information to be available.

References

[Ayers and Völkel, 2008] Ayers, D. and Völkel, M. (2008). Cool URIs for the Semantic Web. *http://www.w3.org/TR/cooluris/*. Accessed 2014-12-22.

[Berners-Lee, 2006] Berners-Lee, T. (2006). Linked Data – Design Issues. *http://www.w3.org/DesignIssues/LinkedData.html*. Accessed 2015-01-01.

[Bizer et al., 2009] Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked Data–The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS), 2009*.

[Bizer and Schultz, 2009] Bizer, C. and Schultz, A. (2009). The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24.

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and Van Harmelen, F. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *The Semantic Web–ISWC 2002*, pages 54–68. Springer.

[Chaudhuri and Narasayya, 1997] Chaudhuri, S. and Narasayya, V. R. (1997). An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, volume 97, pages 146–155. Citeseer.

[Erling and Mikhailov, 2009] Erling, O. and Mikhailov, I. (2009). RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer.

[Gonzalez et al., 2012] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 12, page 2.

[Graefe et al., 2011] Graefe, G., Idreos, S., Kuno, H., and Manegold, S. (2011). Benchmarking Adaptive Indexing. In *Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 169–184. Springer.

[Gupta et al., 2013] Gupta, P., Goel, A., Lin, J., Sharma, A., Wang, D., and Zadeh, R. (2013). WTF: The Who to Follow Service at Twitter. In *22nd International World*

Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, pages 505–514.

[Gurajada et al., 2014] Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M. (2014). TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300. ACM.

[Harris et al., 2009] Harris, S., Lamb, N., and Shadbolt, N. (2009). 4store: The Design and Implementation of a Clustered RDF Store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, pages 94–109.

[Heath and Bizer, 2011] Heath, T. and Bizer, C. (2011). *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool, 1st edition.

[Hector et al., 2009] Hector, G.-M., Ullman, J. D., and Widom, J. (2009). *Database Systems: The Complete Book*. Prentice-Hall.

[Hickson et al., 2014] Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E. D., O'Connor, E., and Pfeiffer, S. (2014). HTML5: A vocabulary and associated APIs for HTML and XHTML. *http://www.w3.org/TR/html5/*. Accessed 2014-12-09.

[Idreos et al., 2007] Idreos, S., Kersten, M. L., and Manegold, S. (2007). Updating a Cracked Database. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 413–424. ACM.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM.

[McBride, 2001] McBride, B. (2001). Jena: Implementing the RDF Model and Syntax Specification. In *SemWeb*.

[Neumann and Weikum, 2009] Neumann, T. and Weikum, G. (2009). The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal, 19(1):91-113, 2010*.

[Prud'hommeaux and Seaborne, 2008] Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF – Grammar. *http://www.w3.org/TR/rdf-sparql-query/#sparqlGrammar*.

[Raggett et al., 1999] Raggett, D., Le Hors, A., and Jacobs, I. (1999). HTML 4.01 Specification. *http://www.w3.org/TR/html401/*. Accessed 2014-12-09.

[Selinger et al., 1979] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. (1979). Access Path Selection in a Relational Database Management

System. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM.

[Stutz et al., 2010] Stutz, P., Bernstein, A., and Cohen, W. (2010). Signal/collect: Graph Algorithms for the (Semantic) Web. In *The Semantic Web–ISWC 2010*, pages 764–780. Springer.

[Stutz et al., 2013] Stutz, P., Verman, M., Fischer, L., and Bernstein, A. (2013). TripleRush: A Fast and Scalable Triple Store. In *SSWS@ ISWC*, pages 50–65.

[Zeng et al., 2013] Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z. (2013). A Distributed Graph Engine for Web Scale RDF Data. In *Proceedings of the VLDB Endowment*, volume 6, pages 265–276. VLDB Endowment.

List of Figures

List of Tables