Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Masterthesis

Partout: A Distributed Approach Towards Scalable RDF Processing

---

Submitted by

Luis Antonio Galárraga Del Prado

February 28, 2012

Supervisors
Katja Hose, Max-Planck Institute for Informatics, Saarbrücken
Ralf Schenkel, Max-Planck Institute for Informatics, Saarbrücken

Reviewers
Katja Hose & Ralf Schenkel

UNIVERSITÄT
DES
SAARLANDES

Eidesstattliche Erklärung

Ich erkläre an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, February 28, 2012 .....

(Datum / Date) (Unterschrift / Signature)

Abstract

The increasing interest in Semantic Web technologies has led not only to a rapid growth of semantic data on the Web but also to an increasing number of sources and backend applications with more than a trillion triples in some cases. Confronted with such huge amounts of data, existing state-of-the-art systems for storing RDF and processing SPARQL queries will be no longer sufficient. This thesis introduces PARTOUT, a distributed engine for scalable RDF processing in a cluster of machines. It shows how to fragment RDF collections based on a query log, how to allocate the fragments to nodes in the cluster, and how to efficiently process SPARQL queries. Experiments with several large collections including a standard benchmark, show the superiority of PARTOUT over naive partitioning techniques applied in today's distributed SPARQL engines.

Acknowledgements

I would like to thank all the people who made possible to conclude this work.

My eternal gratitude to my supervisors Dr. Katja Hose and Dr. Ralf Schenkel for their outstanding guidance, engagement and support during all the stages of this thesis. It has been an honor and a pleasure to work with them.

Special thanks to the family Cronshaw: Aracely, Keith, Lynn and Kevin for their inestimable help and affection when I came to this distant place. Thanks for letting me be part of your life.

I cannot forget my good friends who were always concerned about my progress and whose support was crucial during the last few months, specially in the tough moments.

And last but not least, thanks to my dear parents Edwin Galárraga and Patricia Del Prado who despite the distance, were always encouraging me to follow my dreams. I especially dedicate this work to my beloved mother, my friend and mentor, whose legacy lives in every accomplishment of my life.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The increasing interest in Semantic Web technologies has led to a rapid growth of available semantic data on the Web. Recent advances in information extraction [20, 39, 55, 60] make it possible to extract knowledge from natural language text in an efficient and accurate way and represent it in a machine-readable format, with RDF (Resource Description Framework) as the most popular nowadays. Initiatives like the DBpedia[1] project support this assertion. At the moment of writing this thesis (January 2012) , the latest version of the project (DBpedia 3.7), has reached a size of 3.64 million entities and approximately 1 billion RDF facts extracted from Wikipedia. As the number of Wikipedia articles increases every day and information extraction techniques keep improving, DBpedia and similar knowledge bases are likely to keep on growing and require more and more resources for retrieval, processing and storage. Moreover, The W3C has confirmed the existence of commercial data sets which have already exceeded the 1 trillion triples barrier[2].

This growing trend has been the motivation for other initiatives like the Semantic Web Challenge[3] which tries to push the state of the art in Semantic Web, towards a better exploitation of the current plentifulness of semantic data and sources. The 2011 challenge required participants to design and implement an innovative software system on top of a two billion triples dataset.

But it is not only the amount of data provided by a source that is increasing; also the number of sources as the steady growth of the Linked Open Data (LOD) cloud[4] [6] in Figure 1 shows. By September 2011, the LOD cloud consisted of 295 datasets[5] which add up to approximately 30 billion triples [6] LOD sources such as DBpedia interlink their data by explicitly referencing data (URIs) provided by other sources and therefore building the foundations for answering queries over the data of multiple sources.

Furthermore, more and more small RDF data sets without query processing interfaces become available on the Web. The data of such sources can usually be downloaded and processed locally.

Query processing in these scenarios is challenging because of the different ways in which sources can be accessed. Some sources provide SPARQL endpoints, others are available as downloadable data dumps, and still others as dereferenceable URIs,

[1] http://dbpedia.org/
[2] http://www.w3.org/wiki/LargeTripleStores
[3] http://challenge.semanticweb.org/
[4] http://linkeddata.org/
[5] http://www4.wiwiss.fu-berlin.de/lodcloud/state/#domains
[6] http://www4.wiwiss.fu-berlin.de/lodcloud/

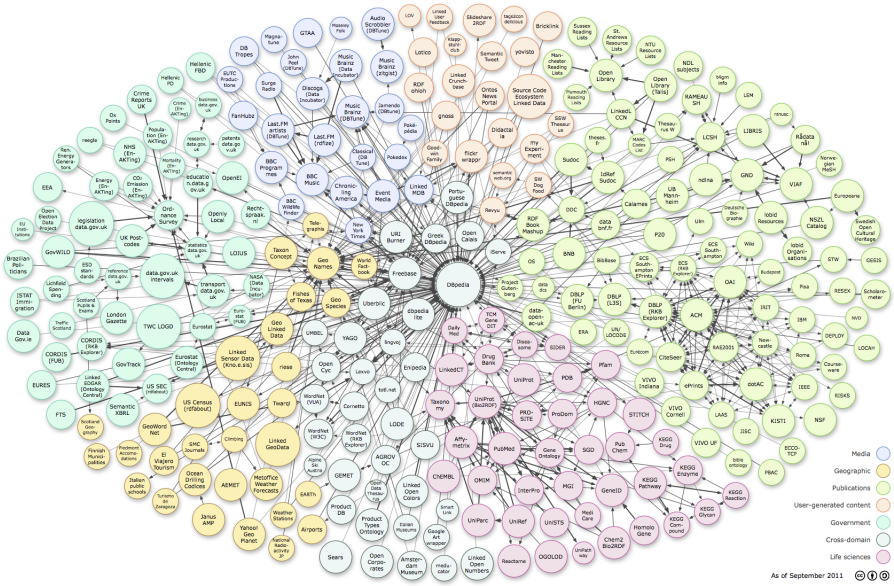


Figure 1: Linking Open Data cloud as of September 2011.

which means a HTTP lookup of the URI will provide a set of RDF triples with facts describing the entity identified by the URI. This led to a variety of approaches for query processing that ranges from downloading the data at processing time [27, 29, 30, 34, 35] to the application of techniques known from distributed database systems [4, 36, 51, 58] since a set of SPARQL endpoints resembles a mediator-based or federated database system [21].

At the other end of the spectrum is data warehousing, where the data is downloaded from the original sources and collected in a huge triple store. Query processing in such a setup benefits from efficient centralized query optimization and execution.

1.1 Problem Statement

All the aforementioned approaches to address scalable query processing have strengths and weaknesses. Federated solutions where every data source is independent, in general avoid data maintenance and storage costs at the price of complex query processing which normally leads to high response times in comparison to centralized approaches. Data warehousing, on the other hand, suffers from the need to update and recrawl the data regularly which in combination with an steady data growth trend will sooner or later result in scalability problems. However, there exist in general two common approaches to address scalable data processing in data warehouses: buying bigger and expensive mainframes that can hold and process most of the data in main memory (centralized processing) or use a scale-out architecture based on

commodity hardware where multiple cheaper machines cooperate to achieve the same goal (distributed query processing in a cluster).

Unlike fully centralized approaches, scale-out architectures have an inherent scalability nature because the capacity of the system can be easily extended by simply adding more machines to the infrastructure. But in order to implement such kind of architecture, we need to address two major problems: (i) partitioning the data in an optimal way plus assigning the partitions to hosts (fragmentation and allocation) and (ii) guarantee efficient distributed query processing.

The concept of optimal fragmentation and allocation is context-dependent. Semantic data is always consumed by software applications which normally have predefined access patterns, thus an optimal fragmentation scheme should optimize for such patterns in order to provide two guarantees: load balancing and scalability. These guarantees are strongly interrelated because a skewed distribution of the load will always lead to scalability problems as the overloaded components will become a bottleneck during execution. In other respects, efficient query processing normally implies small response time, high througput and good resource utilization, which again are related to our previous guarantees because a system with scalability problems will hardly achieve those goals.

1.2 Contribution

This thesis explores the use of a scale-out architecture for scalable and efficient processing of RDF data. In summary, it provides the following contributions:

- PARTOUT[7], a distributed engine based on a scale-out architecture for scalable RDF processing and distributed SPARQL query processing in a cluster of machines,
- a novel data partitioning and allocation algorithm for RDF data in consideration of a given sample query load, and
- a distributed query processing strategy and a cost model for the proposed architecture.

Our evaluation shows that our approach is superior over the naive partitioning techniques applied in state-of-the-art distributed SPARQL engines.

This thesis is structured as follows. After having discussed related work and preliminaries in Section 2, Section 3 presents a novel method for query load aware

[7] pronounced like the French word *partout*; the name is a combination of the words *part*ition and scale-*out*

RDF partitioning and allocation. Section 4 then presents algorithms for efficient query processing and optimization. Section 5 shows the most relevant implementation details and challenges faced to build PARTOUT. An evaluation of the proposed approach is covered in Section 6. Finally, Section 7 concludes the thesis with a summary and an outlook to future work.

2 Background and Related Work

In this section, we present background concepts relevant to the context of semantic data, efficient query processing and database fragmentation and allocation. First we introduce the idea of the Semantic Web and two of its building blocks: the RDF data format and the SPARQL query language. Then, we address the different alternatives for efficient and scalable query processing to finally discuss about fragmentation and allocation for databases.

2.1 The Semantic Web

Coined by Tim Berners-Lee, the inventor of the World Wide Web, the Semantic Web can be defined as an evolutionary stage of the current WWW, where the web of documents evolves into a web of data. Web pages are pieces of more or less structured information written in natural language, tailored for humans but hard to process by computers. In contrast, in the web of data, the information is in a format easy to read by computers. This provides a huge potential for sophisticated software agents relying on efficient information search and retrieval with the possibility of even more powerful inference approaches and easy data integration (by linking). The document is not anymore the central unit of information, but it is instead replaced by the **resource**. In this context, a resource is a conceptual model of any real-world entity like a person, a book or a place. As Tim Berners-Lee stated in the original article introducing the Semantic Web [62], "It is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation".

2.1.1 Knowledge databases

Giving information a well-defined meaning implies it has to be represented in a format that allows computers to "understand" the semantics of the data. Unlike the web of documents where those semantics are provided implicitly, the web of data requires them to be explicitly expressed. This allows us to introduce the concept of an **ontology**: the description of the concepts and relationships that can exist for a group of resources. If our resources are people, then an ontology can explicitly state that people can be friends of other people. This concept of friendship between two resources of type person is what an ontology describes. The popularity of the semantic web initiative has led to the emergence of many ontologies for a broad range of domains like Media, Life Sciences, Goverment among others. Popular ontologies

like FOAF (Friend of a Friend)[8], BIO (A vocabulary for biographical information)[9] or Dublin Core[10] are frequently used across domains. Moreover, the W3C offers two recomendations for the specification of ontologies. RDF Schema[11] which provides basic elements for the description of ontologies, whereas the OWL Web Ontology Language[12] and its three sublanguages OWL Lite, OWL DL and OWL Full offer a wider and incremental set of features to design ontologies.

2.1.2 RDF - Resource Description Framework

Besides the framework provided by an ontology, the web of data requires a format to represent the actual facts or knowledge. There is where RDF (Resource Description Framework) comes into play. It is a language aimed for representing information or knowledge about resources and is a W3C recommendation[13]. It is one of the building blocks of the Semantic Web.

RDF represents pieces of knowledge as triples of subject, property and object. The subject is a resource with a unique identifier. It is the real world entity, this unit of knowledge talks about. The property, on the other hand, is the quality or trait of the resource that is being described, whereas the object is the value associated to that property.

Example 1. An RDF triple

```
@prefix db:<http://dbpedia.org/resource/>
@prefix dbp:<http://dbpedia.org/property/>
db:The_Lord_of_the_Rings dbp:title "The Lord of the Rings"@en
```

Example 1 shows a typical RDF triple using the N3 format [14]. Subjects in RDF are always resources which can be identified either by a URI or a blank node. A blank node is a resource with an identifier with limited scope, normally the file or database where the triple is stored in contrast to URIs which are designed to have a global scope and allow for external linking. Properties are always URIs. Objects can be either URIs, blank nodes, or literals. Literals are used to identify values such as numbers, strings, or dates by means of a lexical representation. A literal can be plain or typed. A plain literal is a string combined with an optional language tag and they are normally used for plain text in a natural language like in Example 1.

[8] http://www.foaf-project.org/
[9] http://vocab.org/bio/0.1/.html
[10] http://dublincore.org/
[11] http://www.w3.org/TR/rdf-schema/
[12] http://www.w3.org/TR/owl-features/
[13] http://www.w3.org/RDF/
[14] http://www.w3.org/TeamSubmission/n3/

A typed literal is a string with the lexical representation of a value of a particular datatype, whose URI is always explicitly written. The lexical representation of a value is its string representation. For example the strings “true” and “1” are two lexical representations for the boolean value *true*. The N3 keyword `@prefix` can be used to declare abbreviations for prefixes and is useful when many URIs in a dataset contain the same prefix so that redundancy is avoided. In Example 1, `dbp:title` is actually an abbreviation for `http://dbpedia.org/property/title`

Example 2. RDF Triple with a typed literal in the object position

```
@prefix db:<http://dbpedia.org/resource/>
@prefix dbo:<http://dbpedia.org/property/>
db:Abraham_Lincoln dbo:deathDate "1865-04-15"^^<http://www.w3.org
    /2001/XMLSchema#date> .
```

Even though RDF databases can be seen naturally as triple stores, they are studied frequently using the graph based data model as Figure 2 shows. The web of data is about representing knowledge, but also about linking it with other knowledge provided by independent sources. For that reason, a semantic database can be seen as a directed and labeled graph, which we call the *Data Graph*, where subjects and objects are nodes and the predicates are represented by directed, labeled edges. This model is normally useful for formal specification, as well as for visualization because it provides an insight of the actual data topology.

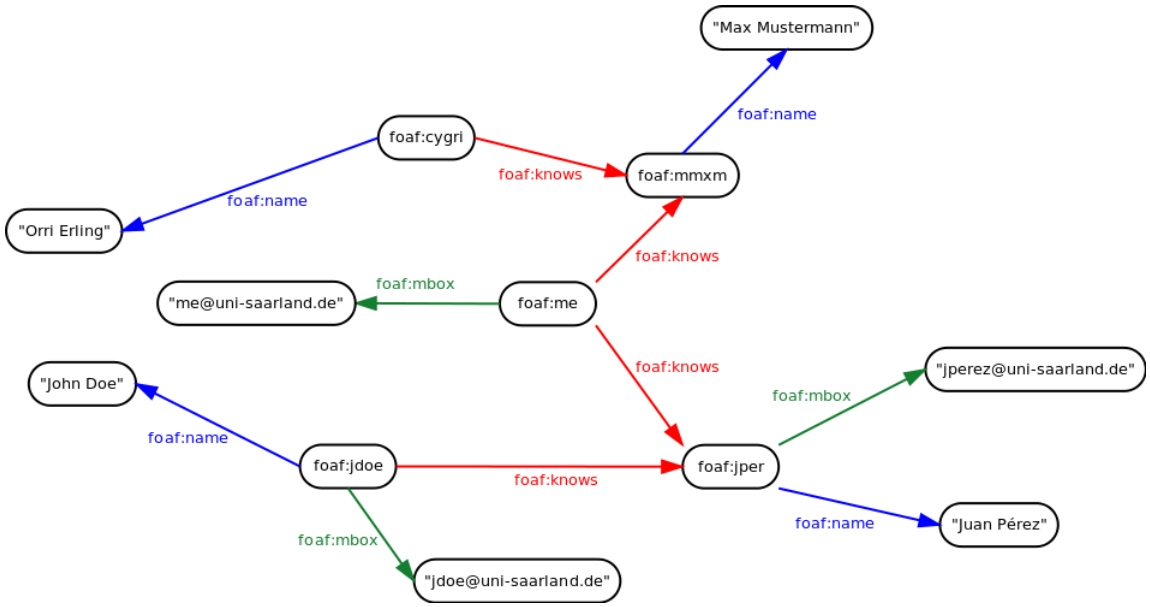


Figure 2: An RDF data graph

With this introduction, we are ready to provide the formal definition for a triple store, based on some definitions given by the W3C [57]:

Definition 1. *Let U, B and L be pairwise disjoint infinite sets of URIs, blank nodes and literals. An RDF triple t (subject, property, object) is an element of $(U \cup B) \times U \times (U \cup B \cup L)$. Define $\pi_{subject}(t)$, $\pi_{property}(t)$, $\pi_{object}(t)$ as the subject, property and object values for the RDF triple t.*

Definition 2. *An RDF triple store T is a set of RDF triples. We denote as G_T the corresponding data graph representing T.*

Definition 1 formalizes the definition of an RDF triple and borrows some notation from relational algebra to refer to its components. Then it is used to define an RDF knowledge base or triple store in Definition 2.

2.1.3 The SPARQL Query Language

The SPARQL query language is to RDF Graphs as SQL is to relational databases. It is a declarative language to extract data from RDF graphs and is a W3C standard recommendation[15]. A SPARQL query can be seen as a subgraph pattern that has to be matched against an RDF data graph in order to provide some specific information. It consists of a set of triple patterns. A *triple pattern* has the same structure as an RDF Triple, but it can contain unbound values or variables which are normally prefixed using the question mark character (?). SPARQL queries can be of type SELECT, ASK, CONSTRUCT and DESCRIBE depending on the type of information required from the data graph and its intended use. For example a SPARQL SELECT query is used to retrieve a set of values matched by a subgraph pattern in the data graph, whereas a SPARQL ASK query simply tests whether a given subgraph pattern has a solution or not. It is possible to derive new data graphs with the information matched by a subgraph pattern, by means of a SPARQL CONSTRUCT query which additionally requires a graph template for the structure of the new data graph. SPARQL DESCRIBE queries on the other hand, return subgraphs containing information about the resources matched by the given subgraph pattern. Since a complete formal specification for the SPARQL query language is far beyond the scope of this thesis, we restrict the description only to SELECT queries which are relevant for our purposes.

Definition 3. *Let V be the set of all possible query variables, and define $\Psi := U \cup B \cup L$ with $\Psi \cap V = \emptyset$. A triple pattern p is an element of $(U \cup B \cup V) \times (U \cup V) \times (\Psi \cup V)$.*

[15] `http://www.w3.org/TR/rdf-sparql-query/`

Moreover, let $\pi_{subject}(t)$, $\pi_{property}(p)$, $\pi_{object}(p)$ *be the subject, property and object components of the triple pattern* p.

Definition 4. *A basic graph pattern is a set of triple patterns.*

Definitions 3 and 4 are the building blocks for any SPARQL query and provide the basis to introduce the concept of a *SPARQL expression*.

Definition 5. *A SPARQL expression is recursively constructed as follows. (1) A basic graph pattern is an expression. (2) If* Q_1, Q_2 *are expressions and* R *is a filter condition, then* Q_1 $FILTER$ R, Q_1 $UNION$ Q_2 , Q_1 $OPTIONAL$ Q_2 *and* Q_1 AND Q_2 *(the same as* Q_1 . Q_2*) are also expressions.* R *is a SPARQL filter condition* [16] *over* $vars(Q_1)$*, where* $vars(Q_1)$ *is the set of variables occurring in* Q_1.

Example 3. *A SPARQL expression*

```
?person foaf:name ?name .
?person foaf:mbox "me@uni-saarland.de" .
?person yago:birthDate ?bdate .
OPTIONAL { ?person foaf:friend foaf:Fred_Flinstone }
FILTER bdate >= "1985-06-12"^^xsd:date
```

Definition 5 introduces a SPARQL expression, the central component of any SPARQL query and Example 3 illustrates how a basic graph pattern with three triple patterns is combined with other subexpressions to produce a SPARQL expression. However this definition still does not provide a clear insight about what it means to answer a SELECT query in a triple store. For that purpose we need to introduce the concept of a *solution mapping* (Definition 6).

Definition 6. *Let* V *be the set of all possible query variables, and define* $\Psi := U \cup B \cup L$ *with* $\Psi \cap V = \emptyset$*. A solution mapping* μ *is a partial function* $\mu : V \rightarrow \Psi$*. The domain of* μ*, denoted by* $dom(\mu)$*, is the subset of* V *where* μ *is defined.*

The solution of a SPARQL expression over a RDF triple store T is described by a set of mappings between the variables and the constants in the triple store, where each mapping represents a possible answer. In Example 4, $\mu_1(Q)$ and $\mu_2(Q)$ replace the variables with the values in the mappings, producing RDF Triples which must be in T.

Example 4. Consider a SPARQL expression Q:

```
?city yago:label ?name .
?city yago:locatedIn yago:Germany
```

[16] For a formal definition of filter conditions, please refer to [57]

and the mappings μ_1 := {?city → yago:Berlin, ?name → "Berlin"@de}, μ_2 := {?city → yago:Hamburg, ?name → "Hamburg"@de} Then dom(μ_1) = dom(μ_2) = {?city, ?name} and $\mu_1(Q), \mu_2(Q) \in T$

Given the definition of a SPARQL expression, it is straightforward to define a SPARQL SELECT query by simply adding a projection over the variables occurring in the SPARQL expression. This is formalized in Definition 7.

Definition 7. *Let Q be a SPARQL expression and let $v \subseteq V$ be a finite set of variables. A SPARQL SELECT query is an expression of the form $SELECT_v(Q)$.*

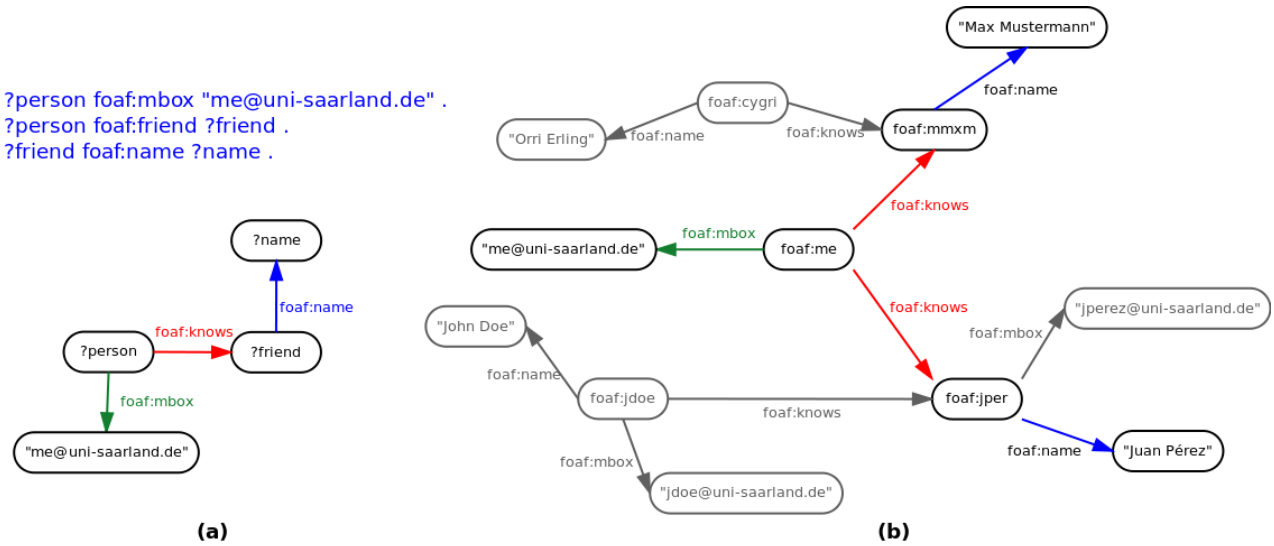


Figure 3: (a) A basic graph pattern (b) A data graph matched by the basic graph pattern in (a)

Example 5. List the names of all the acquaintances of the person whose email address is me@uni-saarland.de

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name WHERE {
    ?person foaf:mbox me@uni-saarland.de .
    ?person foaf:friend ?friend .
    ?friend foaf:name ?name .
}
```

Example 5 shows a SPARQL SELECT query which projects one of the variables from its SPARQL expression. Its solution consists of all the values of variable ?*name* across the different mappings in the solution of the SPARQL expression. Additionally, this process can be a seen as a graph matching problem, where the basic graph

pattern contained in the expression is a subgraph that must be matched against the data graph G_T [31] as Figure 3 illustrates.

Finally, Table 1 maps the most relevant features of the SPARQL query language to their RA (Relational algebra) counterparts. It is important to mention that two triple patterns with a variable in common expresses a join operation, therefore we can conclude that queries over RDF data graphs are mostly conjunctive and join oriented.

Table 1: Mapping from a subset of SPARQL to Relational Algebra. Q, Q_1 and Q_2 are SPARQL expressions.

SPARQL Construction	*Operation*	*Expression in RA*
$SELECT_v(Q)$	Projection	$\pi_v(Q)$
Q_1 . Q_2	Join	$Q_1 \bowtie Q_2$
$Q\ FILTER\ R$	Selection	$\sigma_R(Q)$
$Q_1\ UNION\ Q_2$	Union	$Q_1 \cup Q_2$
$Q_1\ OPTIONAL\ Q_2$	Left Outer Join	$Q_1 ⟕ Q_2$

2.2 RDF Query Processing

This section explores the state of the art in RDF data processing. It first mentions several observations made on real RDF triple stores and then discusses the different approaches for RDF processing which depend on whether the data is materialized or virtually integrated from independent sources. It leads to a wide variety of approaches from fully centralized solutions (data warehousing) to parallel and distributed infrastructures. The section concludes with a general overview of the goals and challenges of query processing in information systems.

2.2.1 RDF Stores

The plethora of semantic datasets has motivated an increasing amount of research in the field of query processing on RDF stores. Even though RDF data can be easily represented in relational databases and take advantage from years of research in this field, its schema free nature allows for completely native solutions like RDF-3X [43,44] or Hexastore [65]. But not only the data structure of RDF differs from relational data, also its access patterns. SPARQL queries are often much more join oriented than queries for relational data, hence requests with 15 or 20 joining triple patterns are not rare. This certainly imposes a need for efficient join processing methods which are normally complemented with heavy indexing approaches on the three attributes.

Heavy indexing however, assumes updates are infrequent since any modification to the data implies to update all the indexes, a very expensive operation. This looks intuitive if we consider the fact that semantic data was aimed since its inception for consumption (mainly reads) rather than for online transaction processing (OLTP loads rich in reads and writes). Additionally, semantic data is often generated as the result of an information extraction process which tends to produce new data, instead of updating the existing one. [24] makes some assumptions about RDF data graphs based on observations from real-world datasets. In general values appearing as properties in a triple have much lower selectivity than values appearing as objects, with subjects as the most selective. The rationale behind this observation has its roots in the original purpose of RDF, a language to describe information about resources. The number of different resources in a huge dataset is indeed big, thus any query asking for the triples encoding information about a single resource is highly selective. All these assumptions were taken into account when designing PARTOUT.

2.2.2 Centralized Approaches

Building efficient databases for storing and querying RDF on a single machine has been a very active research topic for some years; Sakr and Al-Naymat [54] give an extensive overview of recent results in the field. As RDF data consists of triples, one approach is to consider the data as a big table with three columns (subject, property and object), where each row represents one RDF statement. Alternatively, the data can be split up into multiple relations, e.g., using different relations for different properties. In both cases, relational database systems can be used to manage the data [54].

The straightforward layout of the RDF data allows for optimization techniques that are usually too expensive when applied in the more general context of relational databases, e.g., heavy indexing on all three attributes and aggregate indexes, which are implemented in native RDF triple stores such as RDF-3X and Hexastore, in contrast to solutions like Jena [9], Sesame [7] or Virtuoso [22] which provide a framework for building semantic applications independent from the underlying storage model, e.g relational, object-oriented, XML, etc. It should be emphasized however that these applications are also frequently used in the context of data integration of remote heterogenous sources.

Property tables [66] provide an alternative storage scheme; a property table combines multiple properties with the same subject so that a single table row may store multiple RDF statements (triples). They can be implemented as materialized views in combination with a partitioning by property scheme in order to cluster the

bindings for triple patterns that join frequently and improve query processing. As shown, for example, by Levandoski and Mokbel [37], in some cases property tables can be more efficient than other storage techniques.

As an alternative to considering triples as rows in relational tables, column stores like SWStore [1] and MonetDB [59], split up the data column-wise, i.e., splitting triples up into separate columns for subject, predicate, and object.

2.2.3 Distributed Approaches

As more and more RDF knowledge bases become available on the Web, research not only has to consider the problem of centralized management but also has to solve the problem of distributed query processing, i.e., how to efficiently answer queries over multiple independent RDF sources. Ideally, all sources provide SPARQL endpoints so that the problem can be solved by adapting techniques developed in the context of distributed database systems [4, 36, 51, 58]. However, there is also a high number of data sources whose data can only be accessed by downloading RDF dumps or by dereferencing URIs. Dereferencing a URI means to perform an HTTP lookup of the URI, which provides a set of RDF triples with facts about the entity identified by the URI. There are basically two ways to handle this kind of distributed data and answer queries. First, find all possibly relevant sources, download the data (crawling), and collect them in a data warehouse for future queries. Or second, identify relevant sources for a given query and download the data during query processing. In the first case, we can use the above mentioned centralized approaches as everything is stored in a centralized database (the data warehouse) whereas in the second case we need techniques that identify relevant sources during query processing for a particular query and download only relevant data [27, 29, 30, 34, 35].

Literature has also proposed approaches for RDF processing based on P2P systems [23], e.g., RDFPeers [8], Atlas [33], GridVine [2], RDFCube [38], and MIDAS-RDF [63].

In general, approaches for distributed processing fall into two categories: bottom-up and top-down. In a bottom-up architecture, the data is provided in a distributed way by a number of independent sources. In data integration terminology, this roughly corresponds to virtually integrated systems [21] or, with respect to the subclasses, to federated or mediator-based integration systems. Top-down architectures, on the other hand, assume to be given a big data collection as well as a number of machines and try to partition the data (fragmentation) and assign them (allocation) to a number of (perhaps independent) servers in a way that allows for efficient distributed query processing. Variations in the design can arise depending on the servers' level

of autonomy. PARTOUT is an example of a system that falls into the category of top-down architectures where the servers have no autonomy.

2.2.4 Parallel Processing and Clusters

The degree of independence between the sources plays an important role. In bottom-up architectures, sources retain the highest degree of independence whereas the top-down approach requires a much higher degree of cooperation as the data is assigned to the sources by a global rule or algorithm for fragmentation and allocation. If the sources, or servers respectively, in a top-down architecture are all part of a local cluster, query processing can also benefit from extensive use of parallel processing techniques.

There is a number of commercial systems using clusters of machines for scalable processing of RDF data [10, 25]. Bigdata[17], for instance, uses B+ tree indexes to organize the data, dynamically partitions the B+ tree into index partitions (key-range shards), and assigns the partitions to the servers. Bigdata uses three indexes (SPO, POS, and OSP) to provide efficient access for different types of RDF triple patterns. For SPARQL processing, bigdata uses Sesame 2[18] for query parsing and generating an operator tree but overrides Sesame's query evaluation strategy using the Storage And Inference Layer (SAIL) API. The SAIL implementation transforms the operator tree into custom operators that are optimized, e.g., join order optimization based on selectivity of triple patterns and applying pipelined joins across cluster nodes.

Another system, OWLIM Replication Cluster[19], also allows to increase query throughput using multiple nodes but it does not distribute the data over the nodes in the cluster. On the contrary, the data is replicated at all nodes, i.e., all nodes have all data.

As an alternative to applying techniques known from parallel databases [18], an increasingly large number of systems uses MapReduce [17]-style techniques to efficiently process RDF data in clusters. Examples for such systems are RDFPath [48], PigSPARQL [56], SPIDER [13], and systems recently proposed by Husain et al. [32] and Ravindra et al. [53]. A common disadvantage shared by all of them is the large response time of MapReduce processes.

[17] `http://www.bigdata.com/`
[18] `http://www.openrdf.org`
[19] `http://www.ontotext.com/owlim/replication-cluster`

2.2.5 Fragmentation and Allocation

An important step for setting up a top-down distributed database system is to split up the data (fragmentation) and assign the fragments to peers (allocation). First papers on this topic were published already twenty years ago for relational databases.

With respect to fragmentation, we need to distinguish two basic approaches: horizontal and vertical partitioning. **Horizontal partitioning** [11, 19, 67], depicted in Figure 4 (a), means that a relation is split up row-wise, so tuples are assigned to different partitions. **Vertical partitioning** [12, 14, 26, 40], on the other hand, means that a relation is split up by its columns and therefore its attributes (columns) are assigned to different partitions as shown in Figure 4(b).

Additionally, hybrid approaches are also possible, like combining horizontal and vertical partitioning [3] or what is called **derived horizontal fragmentation**. In the first scenario, a fragment obtained by any of the two approaches is again conveniently partitioned, whereas derived horizontal fragmentation splits a relation based on the fragmentation of another relation when they are frequently joined on a given attribute. The latter mechanism achieves a 1-to-1 join partnership between the fragments which means the join of the two original relations can be expressed as the union of the joins for the pairs of matched fragments in the join graph, which can be processed in parallel. Figures 4 (c) and (d) depict this scenario.

The correctness of a given fragmentation is defined in terms of two criteria: disjointness and completeness. Disjointness means that the data contained in the fragments does not overlap and avoids duplicated information. Completeness in other respects, guarantees there is no data loss after the process, i.e., all pieces of data are effectively assigned to a fragment. The completeness of a fragmentation scheme is formally defined using reconstruction rules. Given a relation R with fragmentation $F_R = \{R_1, R_2 \dots R_n\}$ and key attributes K (set of attributes in the primary key) the reconstruction rules state that the combination of the fragments should produce the original relation. The fragments are combined using the Union or Join operator for horizontal and vertical fragmentation respectively.

$$R = \bigcup_{i=1}^{n} R_i \tag{1}$$

$$R = \bowtie_K R_i \; i \in \{1, 2, \dots n\} \tag{2}$$

Note that in vertical fragmentation, all fragments share the columns in the primary key, since the reconstruction rules need them in order to connect the different fragments a row is split.

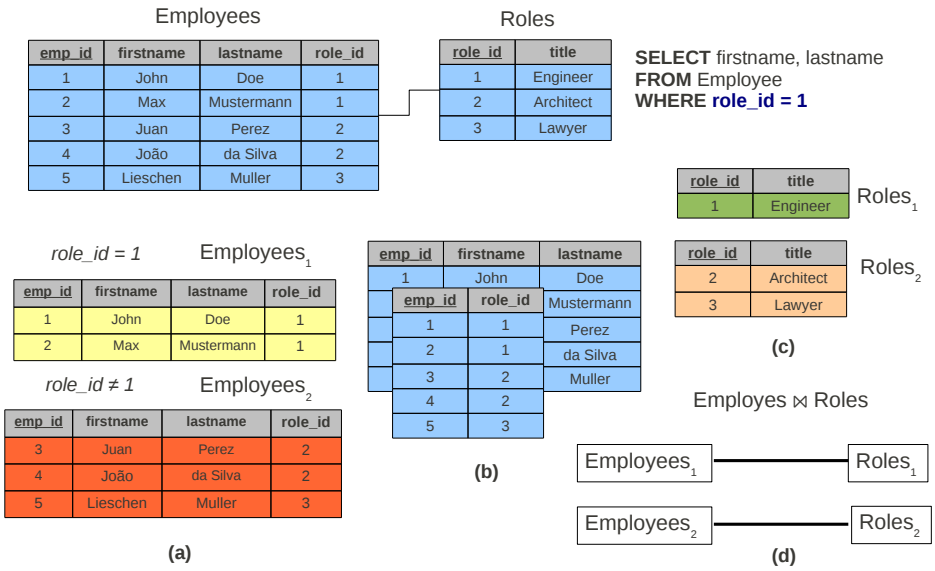


Figure 4: (a) Example of horizontal fragmentation for relation *Employees* based on the boolean predicate $role\_id = 1$ taken from a query. (b) Example of vertical fragmentation for relation *Employees*. (c) Example of derived horizontal fragmentation for relation *Roles* based on fragmentation in (a). (d) Join graph for the operation $Employees \bowtie Roles$ using the fragments.

The purpose of any fragmentation approach is to cluster pieces of data accessed together in the same fragment for the sake of efficient query processing. In some cases it implies queries can run in a single fragment or take advantage of parallelism like in the previous example. This is normally achieved by analyzing the data access patterns, i.e, in a query log and use them accordingly. For horizontal fragmentation, it implies to extract boolean predicates from the queries to partition the tuples set based on whether they match or not those predicates like in Figure 4(a). The optimal horizontal fragmentation algorithm [19, 45] achieves this by using the minimum number of boolean predicates given a big number of predicates extracted from the query load. For vertical fragmentation, capturing the access patterns means to cluster columns based on their co-occurrence in the queries [40].

A recently proposed approach, Schism [15] uses a graph-based method to find a partitioning of relations. Nodes in the constructed graph correspond to tuples, edges connect two nodes when a transaction accesses data of both nodes. After having applied a graph partitioner to split the graph into partitions, a machine-learning technique is applied to find a predicate-based explanation of the partitioning strategy.

Another approach is to use the database system's internal optimizer to obtain

accurate statistics and cost estimations for a given query load. Such an approach was, for instance, proposed in the context of IBM DB2 [52] where candidate partitions are created based on information such as equality predicates contained in the query load. Given a list of candidate partitions for each table, the advisor combines them and uses the internal optimizer to find an optimal combination for the given query load. Similarly, AutoPart [47] was developed in the context of Microsoft SQL Server. It takes into account a set of attributes with a small set of distinct values for horizontal fragmentation and combinations of attributes used in queries for vertical fragmentation. By using the database system's query optimizer, the approach can select the optimal configuration for a given query load. The most recent work in this area was proposed by Nehme and Bruno [41] in the context of Microsoft's SQL Server. This approach is deeply integrated into the system and uses internal representations of query plans and information about their search spaces. The optimization algorithm identifies interesting columns for partitioning, i.e., columns referenced in equality join predicates and any subset of group-by columns. A branch and bound search algorithm tries to find a good fragmentation based on the identified columns.

All the aproaches discussed so far assume the data follow the relational model; however the notion of fragmentation and allocation is applicable to any data model. The current popularity of OSNs (Online Social Networks) has motivated an increasing amount of work towards the development of scalable infraestructures able to support the steady growth of social data and concurrent requests. Because of this scalability requirement, some of them rely on scale-out architectures [49, 50]. The approach implemented in SPAR for example [49] partitions the data in a way that preserves the underlying community structure as much as possible. SPAR considers replication and ensures that the data of all friends of a user hosted on a particular server is co-located on that same server, thereby guaranteeing local semantics of data. This method does not use a concrete query load to extract access patterns, instead it infers them from the social graph.

The problem when trying to apply these approaches to RDF data and SPARQL queries is that they were developed for scenarios with different characteristics, like joins between multiple relations versus many self-joins typical in RDF data, or OLTP (Online Transaction Processing) loads with a large number of short online transactions (reads and writes) with strong requirements of data consistency, versus the OLAP (Online Analytical Processing) nature of RDF query loads, mainly aimed for reading. Additionally, solutions like Schism [15] will suffer from scalability problems since they strongly rely on the data itself to solve the problem of partitioning, rather than in statistics which are much easier to gather (sometimes they are already there) given

the simple structure of RDF data and its read-oriented access patterns. Finally, vertical partitioning approaches cannot really be applied to RDF data since a triple store is a relation with only three attributes.

Literature also proposes a few systems for RDF processing that make use of replication [22,28]. YARS2 [28], for instance, uses multiple servers and stores RDF triples multiple times as key-value-pairs over a set of machines. However, these approaches do not consider any query load for the decision which tuples to replicate, they simply insert a triple multiple times, e.g., by subject, predicate, and object, so that triple can be found efficiently for any given query. In most cases, the optimization goal is load balancing among nodes instead of the minimization of network load. Moreover, replication makes updates more expensive.

Regarding allocation, any fragmentation scheme must be complemented with an allocation policy which assigns fragments to hosts considering two golden rules: data locality, i.e., place the data as close as possible to where it will be used, and load balancing which means all hosts should receive the same (perhaps proportional to their capacity) amount of load in terms of storage space and query requests. This problem can be formulated as a linear optimization problem [16] which is computationally expensive to solve and incurs in high costs if the data keeps changing or growing and reallocation is frequently needed. For that reason, there exist much simpler approaches based on heuristics or greedy methods which still provide good solutions with minimal computational cost [61]. Strategies like the **Non-redudant best fit method**, simply allocate a fragment to the host that uses its data most frequently. The **All Benefitial Nodes** strategy in contrast, models the benefit and the cost of assigning a particular fragment to a host, based on information about the hosts access patterns, I/O and network latency. It considers replication and places a copy of a fragment to any host where the benefit is greater than the costs. Note however that these methods require previous information about the hosts access patterns for the data, which does not apply for some scenarios like data warehousing.

2.2.6 Distributed Query Processing

From a high-level point of view, the goal of any query processing approach is to answer queries as efficiently as possible. The exact notion of efficiency certainly depends on the concrete application and scenario, but it is safe to state that for most systems, they are oriented to provide properties like low response time and high throughput. **Response time** is defined in literature in two ways: as the time elapsed from the initiation of the query until the retrieval of (a) the first result or

(b) the last result [45]. This metric, however, ignores other less obvious factors like the resource utilization, strongly related to a high throughput. The latter property is a measure of the system's performance and it is defined in terms of completed operations per time unit. In the presence of many concurrent requests, the system and its infrastructure resources are challenged, therefore any sort of waste or load imbalance is not acceptable.

In this context the most common resources are storage space, memory usage, I/O, and CPU time. Furthermore, distribution makes things harder and introduces network bandwith. It is a well-known fact in distributed database systems, that communication costs normally dominate the response time with evidences of ratios in the order of 20:1 between transmission over WANs and I/O time [45]. The situation for LANs is less serious, with experiences suggesting ratios of 1.6:1 [46], which still does not allow to disregard the impact of network transmissions in the overall response time. By contrast, minimizing the communication costs might look counterintuitive with our previous suggestions of optimal resource utilization because it implies that distributed query planners should tend to execute subqueries as local as possible, which does not leverage the great potential for parallelization when there are multiple machines. Moreover, subqueries in SPARQL are basically joins of triple patterns which in combination with the aforementioned ideas implies that the bindings for two joining triple patterns should be allocated in the same place, but independent subqueries should run in parallel in different machines.

A balanced and optimal resource utilization becomes even more crucial for scenarios with scalability requirements. Scalability can be required either because the system must handle more requests per time unit or because the amount of data keeps growing. Top-down architectures have an inherent scalability nature because the amount of available resources can be easily extended, by adding more computers to the existing infrastructure. Unfortunately this is not enough to guarantee low response time or high throughput. If the load distribution in the cluster is highly skewed, the overloaded components will become a bottleneck which impedes good performance. In such scenarios, optimal fragmentation and allocation introduced in the previous section, play a central role. They consider two distinct but correlated elements of the problem: the data distribution and its access patterns. For RDF databases, the first element is related to the data graph topology, whereas the second refers to the way the data is normally queried. In general we can assume data is consumed by applications with predefined access patterns which can be used to distribute the load in a fair way among the computers in the cluster.

3 Load-Aware Partitioning

As mentioned in previous chapters, any top-down approach for query processing requires a previous step where the data is partitioned and allocated among the computers in the cluster. The purpose of this chapter is to describe how this step was implemented in PARTOUT by taking into account the data access patterns encoded in a query load. In summary, the query load aware partitioning process takes a SPARQL query load and extracts its most relevant access patterns to turn them into boolean predicates, used to feed an optimal horizontal fragmentation algorithm which outputs a set of fragments. A second stage takes the obtained fragments and allocates them in a set of hosts so that they get approximately the same load. The discussion starts with some ideas and assumptions about the requirements of a query load in order to describe the methods developed to achieve this.

3.1 Query Loads

A query load is a set of queries that are expected to be issued in a running system. They can either be collected from an existing system or estimated from the queries in applications consuming the RDF data. Queries encode access patterns. In the context of RDF and PARTOUT, they are expressed in the SPARQL query language and provide various information like which constant values are common or which triple patterns frequently join or occur together. They also have to be syntactically correct and non-trivial, which means the query processor must produce a query plan and run it in order to get an answer even if the query has no results. If the queries in a query load meet these requirements, we say the query load is *representative*. The following definition formalize the notion of a representative SPARQL query load.

Definition 8. *A representative SPARQL query load $QL(Q_l, F_l)$ over a triple store T consists of a set Q_l of representative SPARQL SELECT queries and a multiset of positive integers F_l where the i-th element $f_{l_i} \in F_l$ denotes the frequency of occurrence of q_{l_i}, the i-th query in Q_l. The size of QL denoted as $sz(QL)$ is $\sum_{1 \leq i \leq |Q_l|} f_{l_i}$*

Table 2 summarizes the most relevant notation used in the remainder of this thesis.

3.2 Extracting Access Patterns from a Query Load

We assume that we are given a representative query load $QL(Q_l, F_l)$ as pairs of SPARQL queries and positive integers denoting the number of times the query has been issued.

V	Set of all possible variables
U	Set of all possible URIs
L	Set of all posible literals
B	Set of all possible blank nodes
Ψ	$U \cup B \cup L$
q	SPARQL query
$B(q)$	triple patterns of q
$C(q)$	non-optional triple patterns of query q
$O(q)$	optional triple patterns of query q
$F(q)$	filter predicates of query q
$QL(Q_l, F_l)$	query load with query set Q_l and multiset of frequencies F_l
$sz(QL)$	size of a query load QL
q_{l_i}, f_{l_i}	i-th query and frequency components of $QL(Q_l, F_l)$
Θ	normalization threshold for constants in a query load QL
Ω	anonymized variable
$\omega(p)$	anonymized version of triple pattern p
$\Phi(QL)$	set of anonymized normalized triple patterns in QL
G	a graph
$G(QL)$	global query graph
$G(QL, M)$	global fragment query graph
$witness(q, p)$	query q is a witness for the anonymized triple pattern p
$witness(q, p, r)$	query q is a join witness for the anonymized triple patterns p and r
$QL^{p,r}(Q_l^{p,r}, F_l^{p,r})$	subset of the query load with the witnesses for pattern p
$QL^{p}(Q_l^{p}, F_l^{p})$	subset of the query load with the join witnesses for patterns p and r
α, β	simple predicates
M'	set of minterms
M	set of minterms for reduced set of simple predicates
m	minterm and the corresponding fragment (long form: T_m)
F	sets of fragments
T	set of triples to fragment and allocate (triple store)
G_T	graph representation of a triple store T
p,r	triple patterns, possibly normalized and anonymized
$vars(p)$	set of variables of a triple pattern p
$f(p)$	frequency of a triple pattern in QL, weight in $G(QL)$
$f(m)$	frequency of a minterm (based on patterns in QL)
$s(m)$	size of m (in number of triples)
$sm(m)$	bitvector representation of minterm m
n	number of hosts in the cluster
h	a host in the cluster
F_h	fragments assigned to host h
T_h	triples assigned to host h
$L(m)$	load induced by fragment m
L	load induced by all fragments
U	uniform load over all hosts
CL_h	current load of host h

Table 2: Notation used in the remainder of the thesis

For every $q_l \in Q_l$, we replace infrequent URIs and literals in triple patterns with variables to avoid overly fine fragmentation of triples, i.e., building fragments that contain only very few triples and that are accessed only by very few queries. As an example, consider the triple pattern *(?x,foaf:mbox,"alice@example.com")* that appears only once in the query load. We replace the object by an artificial variable, yielding the normalized triple pattern *(?x,foaf:mbox,?v)* that captures the general access pattern without fixing a constant object. Constants (literals and URIs) are important for query processing because they are used for data lookup in efficient data structures like indexes. In our previous example, an index on the property column would allow to efficiently search and retrieve all the triples with property value `foaf:mbox`. Nevertheless, they are in general unstable for partitioning because a constant describing a hot topic could be a good discriminator as long as the topic is of global interest. In our example the normalized version captures the general pattern of searching for a resource given its email address. On the other hand, if the literal "bob@example.com" appears in many queries in the query load, we keep triple patterns such as *(?x,foaf:mbox,"bob@example.com")* even though they match only very few triples, but are beneficial for many queries. We never normalize properties under the assumption that queries with variables in the property position are rare. The frequency threshold Θ ($0 < \Theta \leq 1$) is a tunable parameter that determines how frequent a constant must be in order to survive the normalization process. For example, consider a query load QL with size $sz(QL) = 500$ and $\Theta = 0.01$. Denote as $QL_{const}(Q_{lconst}, F_{lconst}) \subseteq QL(Q_l, F_l)$ the subset of the query load where *const* occurs. If the size $sz(QL_{const})$ of such subset is greater or equal than 5, then *const* is not normalized.

For each triple pattern p extracted from Q_l, we further consider its anonymized version $\omega(p)$ where each variable is replaced by the same anonymized symbol Ω. In the example, the anonymized version of the normalized triple pattern *(?x,foaf:mbox, ?v)* is *(Ω,foaf:mbox,Ω)*. We denote the set of all anonymized normalized triple patterns extracted from query load QL as $\Phi(QL)$. Note that more than one original triple pattern in QL can be mapped to the same anomymized pattern. For instance a query load QL containing the triple patterns *(?x,foaf:mbox,"bob@example.com")*, *(?y,foaf:mbox,"alice@example.com")* and *(?x,foaf:knows,?y)* would produce $\Phi(QL) =$ { *(Ω,foaf:mbox,"bob@example.com")*, *(Ω,foaf:mbox,Ω)*, *(Ω,foaf:knows,Ω)* }, assuming the constant "alice@example.com" is infrequent.

The next step is to take into account the join relationships between the anonymized triple patterns. We first define the concept of a global query graph.

Definition 9. *Let p, r be anonymized triple patterns and $q \in Q_l$ a SPARQL SELECT*

query. q is a witness for p, denoted as $witness(q, p)$ iff $\exists\ p' \in C(q) \mid \omega(p') = p$. Furthermore q is a join witness for p and r, denoted as $witness(q, p, r)$ iff $witness(q, p) \wedge witness(q, r) \wedge vars(p') \cap vars(r') \neq \emptyset$.

A query q is a witness for an anonymized triple pattern p if p was derived from one of the triple patterns in q. In other respects, two triple patterns join in a single query if they share at least one variable, therefore the definition of join witness queries allows to extend the join relationship for anonymized triple patterns, where all variables have been replaced by Ω. This notion is used to build the definition of a global query graph for a query load $QL(Q_l, F_l)$.

Definition 10. *The global query graph for the query load $QL(Q_l, F_l)$, denoted as $G(QL)$ is an undirected weighted graph whose vertices are the elements of $\Phi(QL)$. There is an edge between the triple patterns p and r iff $\exists\ q \in Q_l \mid witness(q, p, r)$. Also define $QL^p(Q_l^p, F_l^p) \subseteq QL(Q_l, F_l)$ as the subset of the query load with witnesses for pattern p and $QL^{p,r}(Q_l^{p,r}, F_l^{p,r}) \subseteq QL(Q_l, F_l)$ as the subset with join witnesses for p and r. The weight of an edge, denoted as $w(p, r)$ is $sz(QL^{p,r})$, the size of $QL^{p,r}$. Likewise, the weight of p in $G(QL)$, denoted as $f(p)$ is $sz(QL^p)$.*

The global query graph encodes the join relationships between the anonymized triple patterns extracted from the query load. Its vertices are anonymized triple patterns whereas there is an edge between two patterns p and r if there is at least one query that is a join witness of them. The weight of the edge (p, r) is the sum of the frequencies of the join witness of p and r in the query load. Vertices have also weights which denote the number of witness queries for the anonymized triple pattern defined by the vertex. The global query graph is used by the allocation strategy described in Section 3.4, to determine the best host for a fragment based on its join relationships with other fragments.

Query	Frequency
q_1 = `?s rdf:type db:city. ?s db:located db:Germany. ?s db:name ?n`	2
q_2 = `?s rdf:type db:city. ?s db:located db:USA. ?s db:population ?p`	1
q_3 = `?s rdf:type db:city. ?s db:located db:Germany. ?s db:population ?p. FILTER(?r`$\leq 5 \times 10^6$`)`	2
q_4 = `?s rdf:type db:company. ?s db:located db:Germany.`	1
q_5 = `?s db:name ?c. ?s db:revenue ?r. FILTER(?r`$\geq 10^9$`)`	2
q_6 = `?s db:name "Apple". ?s db:revenue ?r.`	12

Table 3: Example query load

Example 6. Consider a triple store T with information about a total of 5000 entities, 3000 of them cities and 2000 companies and the representative query load $QL(Q_l, F_l)$ described in Table 3. The table contains only the information of triple patterns and filter conditions (used later in Section 3.3). The following statements are true for QL:

- $sz(QL) = 20$
- Q_l corresponds to the column "Query" in the table.
- F_l corresponds to the column "Frequency" in the table.

Assuming a normalization threshold $\Theta = 0.1$ (a constant must appear in at least 2 queries), we replace constants `db:USA` and `db:company` by a variable but keep `db:Germany`, `db:city`, `"Apple"` and 10^9. For instance, the triple pattern `?s db:located db:USA` becomes `?s db:located ?v` where `?v` is a surrogate variable. The next step replaces all variables by the symbol Ω yielding the following list of anonymized triple patterns.

$p_1 =$ (Ω `rdf:type db:city`)

$p_2 =$ (Ω `db:located db:Germany`)

$p_3 =$ (Ω `db:name` Ω)

$p_4 =$ (Ω `db:located` Ω)

$p_5 =$ (Ω `db:population` Ω)

$p_6 =$ (Ω `rdf:type` Ω)

$p_7 =$ (Ω `db:revenue` Ω)

$p_8 =$ (Ω `db:name "Apple"`)

The corresponding global query graph $G(QL)$ is depicted in Figure 5. Additionally the following statements about QL and $G(QL)$ are true:

- q_1, q_2 and q_3 are witnesses for p_1 because they contain the triple pattern $p =$ *(?s,rdf:type,db:city)* and $\omega(p) = p_1$. The weight of p_1 in $G(QL)$ is 5, because the sum of the frequencies of all witnesses for p_1 is 5.
- q_1 is a join witness for the anonymized patterns p_1 and p_2 because q_1 is a witness for both the joining triple patterns $r_1 =$ *(?s,rdf:type,db:city)* and $r_2 =$ *(?s,db:located,db:Germany)*.

- $Q_l^{p_1,p_2} = \{q_1, q_3\}$ defines the set of join witnesses for p_1 and p_2. It means the edge between p_1 and p_2 in the global query graph $G(QL)$ has weight 4, the sum of the frequencies of q_1 and q_3 in the query load.

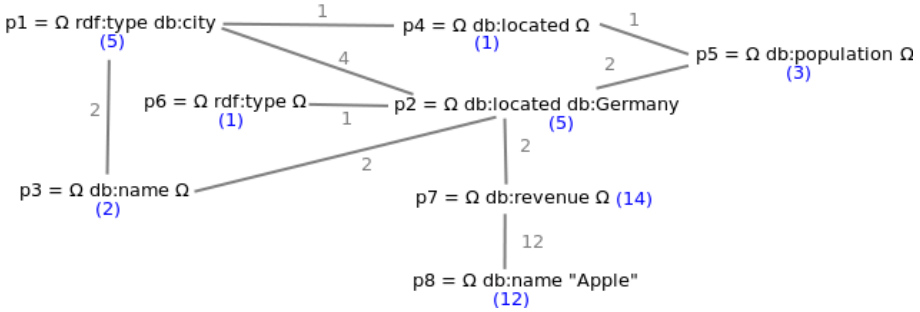


Figure 5: Global query graph $G(QL)$ for Example 6

3.3 Fragmentation

The fragmentation routine leverages the fact that RDF data can be organized in a relation with three columns, so that triples are seen as a set of rows with three attributes: subject, property and object. This allows to apply standard methods for fragmentation in relational databases. These techniques are in general quite efficient and scalable because their complexity does not depend on the size of the data, but on the size of the query load, which is normally smaller. Even though they require some insight about the characteristics of the data, they can rely on summarized statistics which makes them feasible for ever-growing datasets and changing query loads where repartioning is required from time to time.

Given a query load $QL(Q_l, F_l)$, we will now define predicates for splitting a triple store T into disjoint sets of triples. Following standard procedures for horizontal fragmentation of relations [11,45], we will first define *simple predicates* which constrain only one of subject, property, object of triples, and combine them to *minterms*, which are then used to build the set of fragmentation predicates. In the following, we describe the process in detail and provide formal definitions.

Definition 11. *A simple predicate is a boolean expression of any of the following classes:* $a = const$, $a\ op\ const$, $isIRI(a)$, $isLiteral(a)$, $isBlank(a)$, $regex(a, literal)$, $langMatches(lang(a), literal)$, $datatype(a) = uri$ *where* $a \in \{subj, prop, obj\}$, $op \in \{<, \leq, \geq, >\}$, $const \in (U \cup L)$, $literal \in L$ *and* $uri \in U$

Three major classes of simple predicates can be identified from Definition 11. The class of predicates denoted by `a=const` defines an equality test on any of the three

columns with a constant value, either a literal or a URI. Examples of this class are `property=db:located` or `object=db:Germany`. The class `a op const` includes predicates like `object<`10^9`^^xsd:integer`. The latter classes are based on built-in SPARQL functions[20]. Examples are `langMatches(lang(object) = "ES")`, and `regex(object,"sig")`, but not more complex expressions such as `subject=object`. It is important to mention that the latest implementation of PARTOUT supports all the aforementioned classes of simple predicates except `isBlank(a)` and `regex(a,literal)`. Moreover, simple predicates like `object<`10^9`^^xsd:integer` require support for data types. So far, only the basic XML Schema[21] data types like integer, decimal, boolean, string, and double are supported .

To build the set $S(QL)$ of simple predicates for the anomymized query load QL, we consider both the anonymized triple patterns and the filter expressions. For each $t \in \Phi(QL)$, each position that does not contain Ω creates a corresponding simple predicate. For example, when t=*(Ω,bornIn,Germany)*, the simple predicates `property=bornIn` and `object=Germany` are added to $S(QL)$. Similarly, filter expressions are converted to simple predicates if possible like described in Example 6.

Example 7. Consider the anonymized triple patterns extracted in Example 6.

$p_1 =$ (Ω `rdf:type db:city`)

$p_2 =$ (Ω `db:located db:Germany`)

$p_3 =$ (Ω `db:name` Ω)

$p_4 =$ (Ω `db:located` Ω)

$p_5 =$ (Ω `db:population` Ω)

$p_6 =$ (Ω `rdf:type` Ω)

$p_7 =$ (Ω `db:revenue` Ω)

$p_8 =$ (Ω `db:name "Apple"`)

They produce the following simple predicates:

$\alpha_1 =$ `property=rdf:type`

$\alpha_2 =$ `object=db:city`

$\alpha_3 =$ `property=db:name`

$\alpha_4 =$ `object=db:Germany`

[20]For a detailed description of SPARQL builtins, refer to `http://www.w3.org/TR/rdf-sparql-query/#SparqlOps`

[21]`http://www.w3.org/TR/xmlschema11-1/`

$\alpha_5 =$ `object=db:name`

$\alpha_6 =$ `property=db:located`

$\alpha_7 =$ `property=db:population`

$\alpha_8 =$ `property=db:revenue`

$\alpha_9 =$ `object="Apple"`

Additionally, consider the queries with filter conditions in the original query load:

$q_3 =$ `?s rdf:type db:city. ?s db:located db:Germany. ?s db:population ?p. FILTER(?r`$\leq 5 \times 10^6$`)`

$q_5 =$ `?s db:name ?c. ?s db:revenue ?r. FILTER(?r`$\geq 10^9$`)`

They produce the following simple predicates:

$\alpha_{10} =$ `object`$\geq 10^9$

$\alpha_{11} =$ `object`$\leq 5 \times 10^6$.

Example 7 illustrates the process of building the set $S(QL)$ from the anomymized triples patterns extracted from the query load. Filter conditions like in the example are also sources of simple predicates; but some expressions like a comparison between two variables are ignored because they usually involve comparisons between values in different sets of bindings, i.e., more than one triple. Simple predicates are boolean expressions meant to be evaluated on RDF triples. For instance, consider the RDF triple *(db:Berlin,db:located,db:Germany)*. It is clear for this example that the simple predicate `object=db:Germany` evaluates to *true* whereas `property=db:revenue` evaluates to $false$. However, there are triples for which the semantics of some predicates are not properly defined. Consider the simple predicates `object`$\geq 10^9$ and `object`$< 10^9$ and our previous example triple. In spite of being mutually exclusive predicates i.e., they both cannot be true or false for the same triple, it is not clear which of them evaluates to *true* or $false$. In order to deal with these problem cases, ambiguous predicates are combined with other simple predicates to solve the ambiguity. In general, simple predicates of the class `a op const` ($op \in \{<, \leq, >, \geq\}$) are combined with other predicates constraining the data type of the argument. In our example `object`$\geq 10^9$, becomes `datatype(object) = xsd:integer` $\wedge$ `object`$\geq 10^9$. Note that by treating both predicates as a single unit, it is now clear that the triple *(db:Berlin,db:located,db:Germany)* evaluates to $false$, whereas the negated predicate `datatype(object)` $\neq$ `xsd:integer` $\vee$ `object`$< 10^9$ evaluates to *true*.

Since a triple store T can be seen as a relation with three columns, standard techniques for fragmentation of relational databases can be easily adapted in this scenario. Based on the theory of horizontal fragmentation described in [11, 45], given a set of simple predicates, we define the set M' of minterms which consists of all conjunctive combinations of these simple predicates where some may be negated, or formally

$$M' := \{ \bigwedge_{\alpha \in S(QL)} \alpha^* \} \tag{3}$$

where α^* is either the simple predicate α itself or its negation $\neg\alpha$. The number of all possible minterms is exponential in $|S(QL)|$ and when applied to a triple store T, it defines a complete and non-redundant fragmentation of T with each minterm defining a fragment, i.e., a set of RDF triples that match the boolean expression of the minterm. If a fragmentation is complete then every triple in T matches at least one minterm, whereas a non-redundant fragmentation implies that the intersection between two different fragments is empty. Despite the exponential size of M', many minterms actually define empty fragments because they contain contradicting conditions. As an example consider two predicates like `property=db:revenue` and `property=db:name`. It is clear that any minterm containing the positive versions of those predicates is empty, as no triple can have both values as property. Moreover, it is also possible for two different simple predicates to be completely correlated, i.e., they evaluate to true or false for the same subset of triples in T. In that case, one of them is redundant and can be discarded without any impact on the fragmentation defined by M'.

Based on the previous analysis, it is possible to reduce the size of M' by (a) eliminating all minterms that contain contradicting conditions and that are therefore unsatisfiable and (b) omitting redundant (totally correlated) predicates in $S(QL)$. For this purpose we apply the optimal horizontal fragmentation algorithm proposed in [16, 19] which is described next.

In every iteration the algorithm considers a new predicate $\alpha \in S(QL)$ and tests whether its addition makes a difference in the last produced fragmentation scheme (line 8). Note that the initial fragmentation scheme consists of a single fragment containing the whole triple store T. If the addition of a predicate does not change the last fragmentation, it means this predicate is redundant and can be discarded. Otherwise it checks if the previously added predicates became redundant due to the inclusion of α (lines 10-18). Lines 5 and 12 build the set of all minterms for a given set of predicates Q whereas lines 6 and 13 remove unsatisfiable minterms which lead to empty fragments. `property=db:revenue` $\wedge$ `property=db:name` is an

```
1: Q' ← ∅
2: Q ← ∅; F(Q) = {T}
3: for all α ∈ S(QL) do
4:    Q' ← Q ∪ {α}
5:    Determine M(Q') from M(Q)
6:    Remove unsatisfiable minterms from M(Q')
7:    Determine signature F(Q') based on M(Q')
8:    if F(Q') ≠ F(Q) then
9:       Q ← Q'
10:      for all β ∈ Q \ {α} do
11:         Q' ← Q \ {β}
12:         Determine M(Q') from M(Q)
13:         Remove unsatisfiable minterms from M(Q')
14:         Determine signature F(Q') based on M(Q')
15:         if F(Q') ≠ F(Q) then
16:            Q ← Q'
17:         end if
18:      end for
19:   end if
20: end for
21: return M(Q)
```

Algorithm 1: Determines a complete and non-redundant fragmentation of a triple store T using a minimal subset of simple predicates from $S(QL)$

example of such a type of minterms, since both conditions cannot hold for a triple at the same time. Contradicting conditions are very frequent and their identification requires some knowledge about the semantics of functions that may be called. As a second example, consider the minterm `isLiteral(object)` $\wedge$ `isIRI(object)`. This expression is also unsatisfiable because its simple predicates return *true* for disjoint sets of triples. On the other hand, `property=db:revenue` $\wedge\neg$ `property=db:name` is satisfiable.

Since the size of the set of minterms is exponential in the number of predicates, the algorithm does not build it from scratch, but reuses previously constructed minterm sets. Suppose that in a previous iteration the algorithm calculated the set of satisfiable minterms $M(Q_{k-1})$ for the predicates $Q_{k-1} := \{\beta_1, \beta_2, \ldots \beta_{k-1}\}$. When considering the set of minterms for $Q_k := Q_{k-1} \cup \{\beta_k\}$, it builds the new minterms by combining the elements of $M(Q_{k-1})$ with the positive and negative versions of β_k. That leads to $2|M(Q_{k-1})|$ new minterms; however some of them might be unsatisfiable because they contain contradicting conditions, or simply empty because of the distribution of the data. Lines 6 and 13 care about this.

To determine if two fragmentation schemes are equivalent, the algorithm compares

their signatures. The signature of a fragmentation scheme denoted as $F(Q)$ is a sorted vector of $|M(Q)|$ real numbers where each entry is associated to a minterm $m \in M(Q)$ and the value is calculated as the product of $s(m) \cdot f(m)$ where $s(m) > 0$ is the size of the minterm: the number of triples in the triple store T that evaluate to *true*. The access frequency $f(m)$ denotes the number of queries in the query load that access the RDF triples in the fragment. The following definition is aimed to clarify how to calculate $f(m)$.

Definition 12. *Let T be an RDF triple store, $m \in M$ a minterm and $p \in \Phi(QL)$ an anomymized triple pattern. We say that m* overlaps with *p if there is at least one triple in T that matches both m and p.*

This concept is the key to calculate the frequency of access of a minterm and therefore its signature entry. If a minterm overlaps with a triple pattern, then its corresponding fragment contains RDF triples that are in the set of bindings of the triple pattern, thus to answer any query including this triple pattern, the query processor will have to access the triples described by the minterm. Based on this rationale, the access frequency $f(m)$ of m is the sum of the frequencies of triple patterns from $\Phi(QL)$ that overlap with m. Example 8 illustrates the idea.

Example 8. Consider the simple predicates,

$\alpha_1 =$ `property=db:name`

$\alpha_2 =$ `object="Apple"`

the set of minterms generated by them:

m_{11}: `property=db:name` $\wedge$ `object="Apple"`

m_{10}: `property=db:name` $\wedge \neg$ `object="Apple"`

m_{01}: $\neg$ `property=db:name` $\wedge$ `object="Apple"`

m_{00}: $\neg$ `property=db:name` $\wedge \neg$ `object="Apple"`

and the anonymized triple patterns with their frequencies of occurrence in the query load:

$p_1 =$ (Ω `db:name "Apple"`) (12)

$p_2 =$ (Ω `db:name` Ω) (4)

$p_3 =$ (Ω `db:revenue` Ω) (14)

It is clear that p_1 overlaps only with m_{11} whereas p_2 overlaps with both m_{11} and m_{10} since it does not impose any restrictions on the object position. Additionally p_3 overlaps with m_{01} and m_{00}. Therefore, the frequency of access of m_{11} is calculated as $f(m_{11}) = f(p_1) + f(p_2) = 12 + 2 = 16$.

As mentioned before, the size of a minterm, denoted as $s(m)$ is the number of triples in T that match m. $s(m)$ must be greater than zero and could be calculated by evaluating m on each triple in T and counting the ones evaluating to true, but that would be too expensive. Instead, we use statistics about the data and some independence assumptions to estimate the cardinality of combinations of predicates. They are described in detail in Chapter 5.2.

Finally, it is important to remark that the algorithm produces a minimal set of predicates Q and a set of minterms M such that each $m \in M$ defines a non-empty fragment $T_m \subseteq T$ of the set of triples, consisting of all triples that satisfy the minterm predicate m. Q is minimal because it does not contain redundant predicates. Example 9 depicts a scenario where redundant predicates are discarded. Additionally, the obtained fragments form a complete fragmentation of T because any triple is assigned to exactly one fragment by construction of the minterms. For notational simplicity, we will identify fragment T_m by its corresponding minterm m.

Example 9. Consider the simple predicates extracted in Example 7 from the query load in Example 6. The dataset T contains 19500 RDF triples about 3000 cities and 2000 companies.

$\alpha_1 =$ `property=rdf:type`

$\alpha_2 =$ `object=db:city`

$\alpha_3 =$ `property=db:name`

$\alpha_4 =$ `object=db:Germany`

$\alpha_5 =$ `object=db:name`

$\alpha_6 =$ `property=db:located`

$\alpha_7 =$ `property=db:population`

$\alpha_8 =$ `property=db:revenue`

$\alpha_9 =$ `object="Apple"`

$\alpha_{10} =$ `datatype(uri) = xsd:decimal` $\wedge$ `object` $\geq 10^9$

$\alpha_{11} =$ `datatype(uri) = xsd:decimal` $\wedge$ `object` $\leq 5 \times 10^6$.

Moreover, assume all the cities in the dataset have populations up to 4×10^6 inhabitants. In that case, the optimal horizontal fragmentation algorithm determines that α_{11} is redundant because it is fully correlated to α_7. As an example, consider the possible set of minterms produced by $Q := \{\alpha_7\}$. It consists of only two fragments: the triples with property value *db* : *population* and everything else, which we denote as m_1 and m_0. Assuming the triple store contains populations for the 3000 cities in the triple store, $s(m_1) = 3000$. Additionally, $f(m_1) = 3$ because m_1 overlaps with the triple pattern *(Ω,db:population,Ω)* whose access frequency is 3 according to the global query graph in Figure 5. On the other hand, m_0 overlaps with all the other triple patterns yielding $f(m_0) = 38$ and $s(m_0) = 16500$. The signature vector of this fragmentation scheme is the sorted vector $[660000, 9000]$. Now consider $Q := \{\alpha_7, \alpha_{11}\}$ which produces 4 fragments: $m_{00}, m_{01}, m_{10}, m_{11}$. Since α_7 and α_{11} evaluate to true or false for the same set of triples, the signature components of m_{00} and m_{11} are the same as m_0 and m_1. Moreover it also implies that the minterms m_{01} and m_{10} define empty fragments which by definition are not considered for the signature. The addition of α_{11} does not change the signature vector, hence it is redundant and removed by the fragmentation algorithm. This is true even if α_7 and α_{11} are part of more complex minterms containing other predicates. The optimal horizontal fragmentation algorithm produces a minimal set of simple predicates to build a set of non-empty fragments. Table 4 shows the output of the algorithm where the result fragments are shown with their access frequencies (obtained from the global query graph in Figure 5) and sizes. In every row, the symbol ζ is an abbreviation for "all the other predicates are negated" ($\zeta := \neg\alpha_i \wedge \alpha_{i+1} \cdots \wedge \alpha_{|Q|}$)

Table 4: List of non-empty fragments with their sizes and their access frequencies. The last column corresponds to the signature of the fragmentation scheme.

	Minterm	Freq.	Size	Signature
m_1	`property=db:revenue` $\wedge\ \zeta$	14	1970	27580
m_2	`property=rdf:type` $\wedge$ `obj=db:city` $\wedge\ \zeta$	5	3000	15000
m_3	`property=db:name` $\wedge\ \zeta$	2	4499	9998
m_4	`property=db:population` $\wedge\ \zeta$	3	3000	9000
m_5	`property=rdf:type` $\wedge\ \zeta$	1	2000	2000
m_6	`property=db:located` $\wedge\ \zeta$	1	1700	1700
m_7	`property=db:located` $\wedge$ `obj=db:Germany` $\wedge\ \zeta$	5	300	1500
m_8	`property=db:revenue` $\wedge$ (`object`$\geq 10^9 \wedge$ `datatype(uri) = xsd:decimal`) $\wedge\ \zeta$	14	30	420
m_9	`property=db:name` $\wedge$ `obj=Apple` $\wedge\ \zeta$	12	1	12
m_{10}	ζ (Remainder fragment)	0	3000	0

3.4 Fragment Allocation

Once fragments have been defined, they need to be allocated to the n hosts in the cluster. The allocation has two possibly conflicting goals: (1) group fragments that are joined together in queries to the same host in order to avoid communication cost at query processing time and (2) balance the load among hosts. The load is defined in terms of the amount of data assigned to the host as well as the number of queries that are expected to hit it. To define the load $L(m)$ attributed to a fragment m, we consider the number of queries $f(m)$ for which that fragment is accessed, and assume that all $s(m)$ triples of m are accessed for each such query. Consequently, we get $L(m) := f(m) \cdot s(m)$ like in the previous definition for the signature of a fragmentation scheme. Given a fragmentation M, the total load L on our system is therefore

$$L := \sum_{m \in M} L(m) \tag{4}$$

When the load is uniformly balanced over all n hosts in our cluster, each node is assigned a load

$$U := \frac{L}{n} \tag{5}$$

Definition 13. *Given a fragmentation M and query load QL, define the global fragment query graph $G(QL, M)$ as an undirected weighted graph which has all the fragments $m \in M$ as nodes and an edge $\{a, b\}$ if $\exists\, p,\ r \in \Phi(QL)$ such that a overlaps with p, b overlaps with r, and there is an edge $\{p, r\}$ in $G(QL)$; the weight $w(a, b)$ of the edge $\{a, b\}$ is the sum of the weights of all such edges in $G(QL)$ (i.e., the number of queries that contain a join of triple patterns that overlap with a and b, respectively).*

The *global fragment query graph* is built from the global query graph introduced in Definition 10 but unlike that graph, it encodes the join relationships between the fragments and is used to achieve goal (1). Figure 6 shows the global fragment graph for the fragments described in Table 4 using our example query load. Consider fragments 5 and 7. The first one overlaps with the triple pattern *(Ω,rdf:type,Ω)* whereas the second one overlaps with *(Ω,db:located,Germany)*. As those patterns join in the global query graph, an edge between fragments 5 and 7 is added with the number of times the corresponding triple patterns join in the query load. Note also that a triple pattern can overlap with more than one fragment. That is the case of fragments 1 and 8 with the pattern *(Ω,db:revenue,Ω)*.

The allocation problem in our scenario can be formulated as an ILP (Integer Linear Program), described in Appendix A.2, whose solution provides an optimal allocation

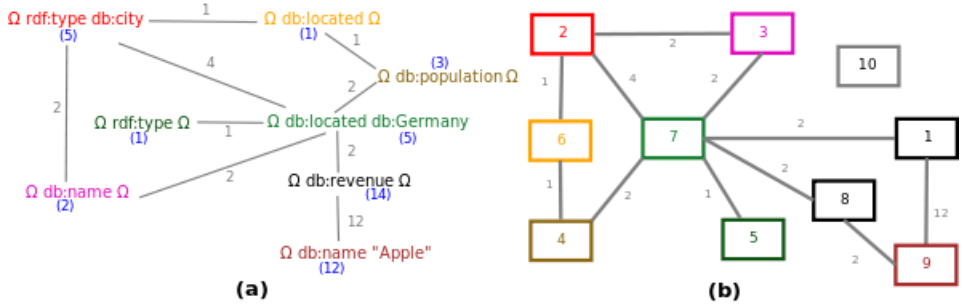


Figure 6: (a) Global query graph from Example 6. (b) The global fragment graph for fragments in Table 4 built from the join relationships in (a). Overlapping triple patterns and fragments have the same color.

of the fragments described by M. Nevertheless, this is a hard problem as ILPs are NP-Complete [5] and therefore infeasible for a large number of fragments and hosts. For this reason, we have designed a straightforward greedy algorithm which runs in $O(n|M|)$ assuming the fragmentation routine outputs the fragments in descending order by their load $L(m)$. Instead of finding an optimal solution, our strategy iterates over each fragment and allocates it to the most promising or beneficial host at that moment based on the aforementioned requirements.

We denote the set of fragments assigned to host h as F_h, and the current load of host h as $CL_h := \sum_{m \in F_h} L(m)$. Initially, no fragments are assigned to any host, so $F_h = \emptyset$ and $CL_h = 0$ for all hosts h.

We can now define the benefit of allocating a currently unallocated fragment m to host h as

$$benefit(m, h) := \frac{2 \cdot U}{U + CL_h} \cdot [1 + \sum_{m' \in F_h} w(m, m')] \tag{6}$$

where U is the expected load each host should receive in a totally fair scenario. The first part of this definition gives higher benefit for allocating a fragment to a host which is currently under-loaded, i.e., its current load CL_h is smaller than U, and reduced benefit for already overloaded hosts. This addresses goal (2). The second part of the definition addresses goal (1) by increasing the benefit when allocating a fragment to a host containing other fragments that join frequently with it. $w(m, m')$ denotes the weight of the edge between fragments m and m' in the global fragment graph. The $+1$ avoids a zero benefit when the host does not contain any fragment that joins with the fragment to be allocated so that the host's current load is not ignored.

For allocating fragments to hosts, PARTOUT uses the following greedy algorithm:

- Fragments are allocated in descending order by their load $L(m)$.

- The first fragment (also the one with the highest load) is assigned to the first host.
- For every fragment except the first one, the benefits of allocating it to every host are calculated. The fragment is then assigned to the most beneficial host.

Example 10 shows the result of allocating the fragments described in Table 4 in 3 hosts.

Example 10. Consider the fragments described in Table 4, their global fragments query graph in Figure 6, and a cluster with 3 hosts, h_1, h_2 h_3. $L(m) = 67210$ so that a perfectly balanced scenario would assign a load of $U = 22403.33$ to each host. The algorithm allocates m_1 in h_1. The next step calculates the benefit of allocating m_2 in every host, since m_1 has been assigned to h_1, its current load is $CL_1 = L(m_1) = 27580$. The calculation for the benefits outputs $benefit(m_2, h_1) = 0.8964$, $benefit(m_2, h_3) = benefit(m_2, h_3) = 2$, therefore m_2 is assigned to h_2. In the same way the benefits of assigning m_3 are $benefit(m_3, h_1) = 0.8964$, $benefit(m_3, h_3) = 2$ but since m_3 joins with m_2 according to the fragment, the formula considers the weight of the edge between m_2 and m_3 which raises the value to $benefit(m_3, h_2) = 3.594$. Thus, m_3 goes to h_2 again. Following the same process for all fragments, the allocation described in Table 5 is achieved.

Table 5: Allocation of fragments to hosts in the example

Host	Fragments	Load
1	m_1, m_9	27592
2	m_2, m_3, m_7, m_8	26918
3	m_4, m_5, m_6, m_{10}	12700

The final step of partitioning must build the actual partitions with the triples from a triple store T. A *partition* is a set of fragments allocated to the same host. The construction of the different partitions implies to scan the triple store to build a global dictionary of strings (i.e., URIs and literals) and a unique mapping of these strings to integer ids, and distribute this mapping to all hosts. At the same time, each RDF triple in T is evaluated against the fragments definitions (minterms) in order to determine the host this triple belongs to. Even though the number of actual fragments is normally far smaller than $2^{|S(QL)|}$, it can be large anyway. If the output of the optimal horizontal fragmentation algorithm determined a minimal set of predicates

Q for fragmentation, then the triple distribution phase takes $O(|T| \cdot |M| \cdot |Q|)$ since in the worst case, the triple has to be evaluated against the $|Q|$ predicates in the definition of every fragment $m \in M$. In order to reduce the amount of evaluations, a definitions summary $summ_h$ is calculated for every host h so that triples are most of the times, only evaluated against this summary. As all minterms are expressed in terms of the same base predicates, define $sm(m)$ as the bitvector representation of minterm m where a 1 at the i-th position, means that the i-th predicate appears in its non-negated version, and 0 means it is negated. We also define a bitwise operator $\bigoplus$ which operates over the bitvector representations of all the fragments $m \in F_h$.

$$summ_h = \bigoplus_{m \in F_h} sm(m) \tag{7}$$

Example 11. Assume a fragmentation and allocation with 4 simple predicates. Further assume that three fragments with bitvector representations 1001, 0001, and 0011 were assigned to certain host. The summary $summ_h$ is:

$$\begin{array}{r} (1001) \\ \bigoplus (0001) \\ (0011) \\ \hline (\ \varnothing 0 \varnothing 1) \end{array} \tag{8}$$

It means the predicates in positions 1 and 3 (from right to left) are decisive and therefore any triple belonging to this host must evaluate to *true* for the predicate at position 1, and $false$ for predicate at position 3. If it does, then the triple is evaluated against all minterms, otherwise it does not belong to this host.

If all the bits at position i have the same value, then the result is that value, otherwise a "don't-care" symbol $\varnothing$ is output. If the ith position of the host summary does not contain $\varnothing$, it means all the fragments allocated to this host contain this predicate in the same form and therefore the predicate is *decisive*. Every triple is then evaluated against the decisive predicates and if it does not match any of them, it can be directly discarded from this host. On the other hand, this approach can produce false positives (e.g a summary containing only $\varnothing$ is the trivial example). In such a case, the triple has to be evaluated against every fragment in F_h to determine whether it belongs to host h or not. Example 11 depicts this scenario.

3.5 Avoiding Imbalance

The construction of minterms based on predicates occurring in queries produces one minterm that contains all predicates in their negated form, i.e., the corresponding fragment contains all triples that do not match any of the simple predicates contained in the queries of the training data set. In many cases, this "remainder" fragment is much bigger than all the other fragments and would lead to a highly imbalanced usage of storage space at the sources. Thus, in case it is significantly bigger than the other fragments, we need to split it up into smaller pieces and assign them to other partitions.

To integrate this consideration into the implementation, all fragments but the remainder fragment are assigned as described in Section 3.4. Afterwards, a hash partitioning is applied to assign the triples of the remainder fragment to partitions. The reason is that, for all the other fragments a global query optimizer can efficiently decide on their relevance for a query by comparing the fragment definition to the predicates contained in the query. In any other case, the optimizer has to use statistics or query all available fragments. In the latter case, the decision whether to ask two or all fragments of roughly the same sizes does not really influence response time because queries are processed in parallel at different hosts whereas querying a much bigger fragment would increase response time. Moreover, other very simple approaches like fragmentation by property are also applicable to this scenario. The remainder fragment could be partitioned by property and the obtained fragments assigned to hosts using PARTOUT's allocation routine with a slight modification in the formula for the load. As the access frequency of these fragments is undefined, the load could be expressed only in terms of the fragment size. The advantage of this method is that untrained triple patterns (assumming they have bound properties) will hit always a single host. However joins of unknown triple patterns will involve many remote transmissions since properties have normally low selectivity and nothing guarantees the joining bindings for such triple patterns will be in the same host. On the other hand, this approach introduces some risks for load balancing because the assignment of fragments to hosts does not consider access frequencies, therefore a host might end overloaded with requests if its assigned properties become popular in the query load. For those scenarios, hash partitioning guarantees a fairer distribution of the load for unknown queries.

4 Distributed Query Processing

Having solved the problem of fragmentation and allocation in the previous chapter, efficient query processing in a top-down distributed setup requires an appropriate global optimizer which given a query in SPARQL produces a good plan according to a cost model. PARTOUT's global optimizer is built on top of RDF-3X [42], one of today's most efficient centralized triple stores whose query planner and cost model are proven to produce efficient execution plans in terms of response time. Nevertheless, RDF-3X' optimizer is not fully applicable to our scenario because it is designed for centralized execution, hence it does not consider communication costs. In this chapter, we briefly describe the RDF-3X architecture in order to introduce our system model and the general setup of software components, and then provide details about our two stages optimization which starts with an RDF-3X plan suitable for centralized execution and then optimizes it for a distributed environment.

4.1 RDF-3X

PARTOUT is built on top of RDF-3X, a native engine for management of RDF data. This system stores triples as a huge relation with three columns S, P, O. In order to provide efficient retrieval, it relies on heavy indexing. There are indexes for every single column (Fully Aggregated) as well as for all the permutations of two (Aggregated) and three columns (Facts). Additionally it uses selectivity histograms to store estimations for the cardinality of joins between triple patterns. Indexes and histograms use unique integers to represent the actual strings and the mapping string-to-integer is stored in a dictionary. Update management is handled by batching operations and incremental indexes that store the modifications before they are merged to the actual indexes. The query processor uses a bottom-up dynamic programming algorithm [44] to find good query plans in terms of response time, and explores both linear deep and bushy trees. Linear deep trees are operator trees with index scans as leaves and binary join operators that have at least one index scan as input, whereas for bushy trees, the inputs of a join can be intermediate results from other operators.

The physical plans use pipelining and implement merge, hash, and nested loop joins. **Pipelining** means an operator will report results to the next operator in the tree hierarchy as soon as it has a result.

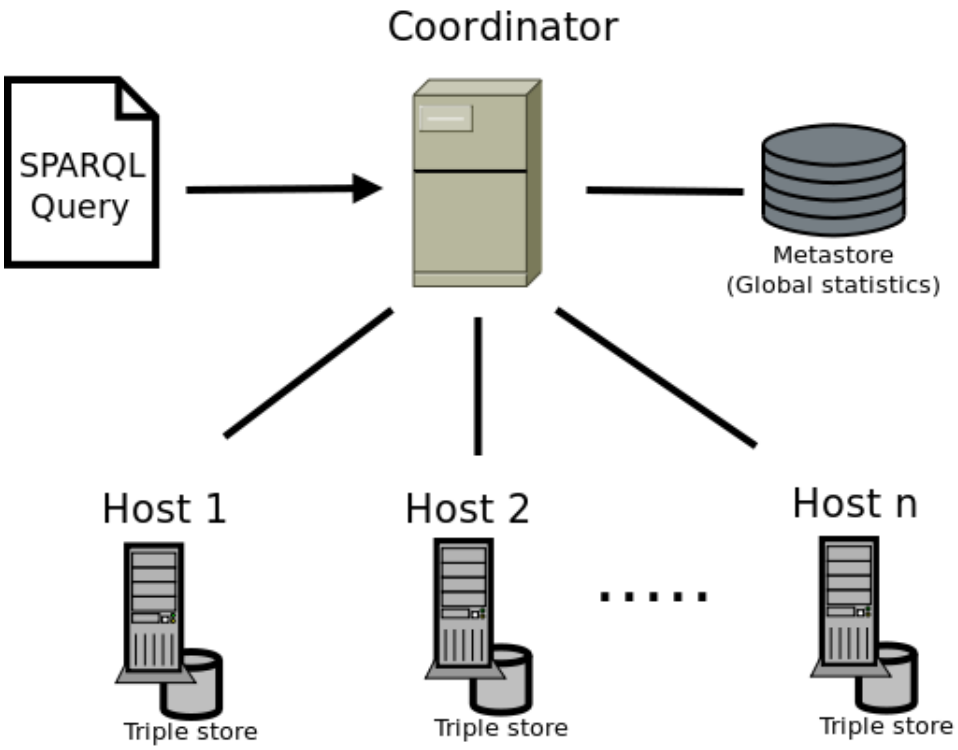


Figure 7: System Model of PARTOUT's query processor

4.2 System Architecture

Figure 7 depicts our system model. Two major components can easily be identified. A central coordinator and a cluster of n hosts or slaves in charge of a subset of the data which we call a *partition*. A partition is the union of all the fragments assigned to a host by the allocation routine. Queries are issued at the central coordinator that stores all relevant information required for query planning like the partitions descriptions, the assignment of fragments to hosts, and data statistics. The slaves are identical process instances whose only responsibility is to execute the subquery plans assigned to them by the coordinator over their subset of the data. The two components have clearly defined and non-overlapping responsibilities and are implemented on top of the RDF-3X engine.

4.2.1 The Coordinator

The coordinator is a program that takes a SPARQL query as input, generates a query plan for the distributed setup, sends it to the slaves, and outputs the query results once they are computed. It does not have direct access to the actual data but instead uses a global statistics file generated at partitioning time, which we call the *metastore*. This is an SQLite database storing all information required for query

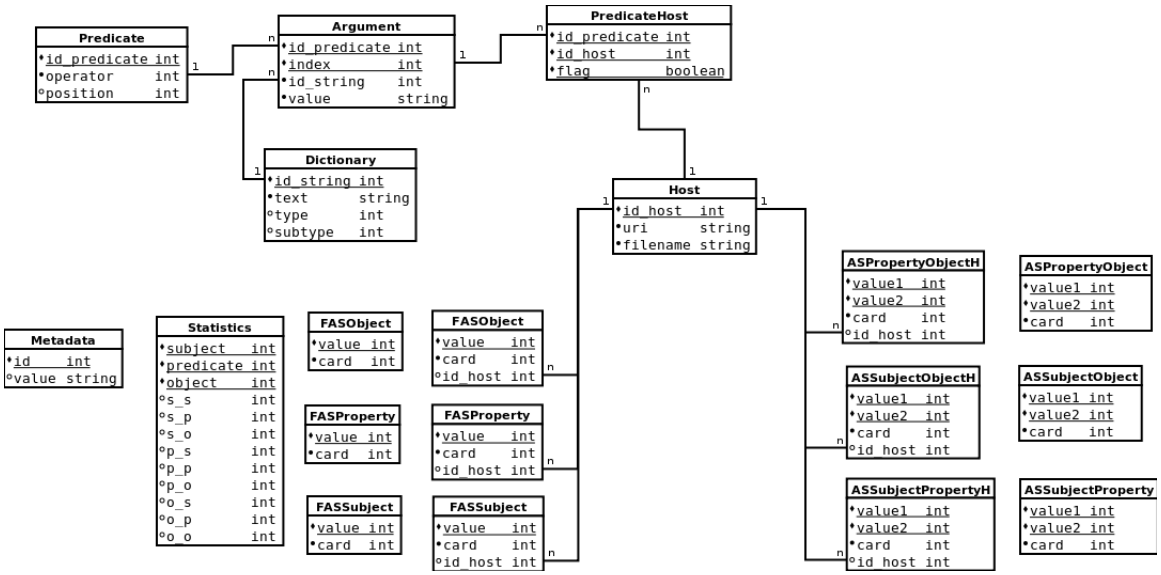


Figure 8: The metastore schema

planning, including:

- *Fragment definitions and host mappings.* Information about how the fragments were created and which hosts they were assigned to.
- *String dictionary.* with the unique string-to-integer mapping used in all hosts.
- *Statistics.* The coordinator stores information about the cardinalities of the normalized triple patterns extracted from the query load and later used for partitioning. It also contains information about the join cardinalities for such triple patterns. These statistics are required by the cost model to estimate the time to evaluate the different operators in the query plan.

Figure 8 shows the relational schema of the metastore. Relations *Predicate*, *Argument*, *Host* and *PredicateHost* encode the fragments allocation information. The first two contain information about the simple predicates output by the optimal horizontal fragmentation algorithm introduced in Chapter 3. A simple predicate like $object = Germany$ would produce a register in the Predicate relation ($position = object, operator = Equal$) , whereas the string *Germany* corresponds to a row in the table *Argument*. The two latter relations store information about the hosts and how the predicates combined as minterms were assigned to them. As an example, consider two hosts h_1 and h_2 and two simple predicates p_1 and p_2 with their four possible fragments: $\{p_1, p_2\}, \{p_1, \neg p_2\}, \{\neg p_1, p_2\}, \{\neg p_1, \neg p_2\}$. Furthermore assume the two first minterms were assigned to h_1. This assignment would produce the following records in the PredicateHost relation: $(p_1, h_1, true), (p_2, h_1, true), (p_2, h_1, false)$ as

p_1 appears always in normal form, but p_2 appears both in its normal and negated forms. Likewise if the remaining minterms were assigned to h_2, it would produce the registers $(p_2, h_1, false), (p_1, h_2, false), (p_2, h_2, true), (p_2, h_2, false)$.

The dictionary table stores the string-to-integer mapping using the same data schema of RDF-3X. Those integers are used everywhere to refer to a string ocurring in the triple store.

The statistics are split among several relations. The tables with prefixes "AS" and "FAS" correspond to the Aggregated and Fully Aggregated Indexes implemented in RDF-3X and store the cardinalities of the normalized triple patterns extracted from the query load. Relations with the suffix "H" like FASPredicateH or ASPropertyObjectH store the cardinalities of the triple patterns per host: FASPredicateH for patterns with only bounded property like *(Ω,bornIn,Ω)* whereas ASPropertyObjectH refers to patterns with bounded property and object like *(Ω,bornIn,Germany)*. This information is precisely obtained at partitioning time. Suppose that for the normalized triple pattern *(Ω,bornIn,Ω)* the partitioning routine found there are 1000 bindings in h_1 and 500 in h_2, the aggregated values for the whole dataset are stored in the view FASProperty as a tuple $((string2integer(bornIn), 1500)$ where $string2integer$ denotes the integer identifier for the string $bornIn$ in the dictionary. In the same way, relations FASSubject, FASProperty, ASSubjectProperty, ASSubjectObject and ASPredicateObject are views aggregating the counts for triple patterns among the hosts.

The *Statistics* relation on the other hand, contains the join statistics for the normalized triple patterns in the whole dataset and uses the same schema as the RDF-3X selectivity histograms to estimate the cardinality of joins [22].

Finally, the *Metadata* relation contains aggregated information about the total number of triples as well as the total number of RDF-3X data pages per type of index (Facts, Aggregated and Fully Aggregated) in the whole system.

It is worth remarking that the metastore includes only statistical information relevant to the triple patterns found in the query load in order to keep the file size as small as possible, i.e., it fits to main memory. Note also that it stores statistical information about both the entire dataset and the partitions because they are necessary for the two steps of the query planning process, first producing a plan aimed for a centralized triple store and then optimizing it for distributed execution. Additionally, if statistical information about a triple pattern unknown to the metastore is needed, the query processor relies on RDF-3X default behavior on the absence of such information, which is an optimistic approach that normally

[22]For more details refer to the section 6.1 of [44]

assumes a value of 1 for the cardinality of triple patterns. However, this fact does not prevent the query processor to gather more statistics from the slaves, as new queries are issued. If the coordinator gets a new triple pattern such that all its derived simple predicates are not in the metastore, it means this triple pattern overlaps with the remainder fragment which is splitted up among all hosts. In order to get an estimation for the number of bindings of such a triple pattern, it will have to ask every host about how many matches they have estimated. Moreover, if the coordinator has contraints in terms of storage space, it could use the metastore schema as a sort of cache for relevant information like statistics and string mappings. For example, it could initially store the string-to-integer mappings only for the constants ocurring in the query load and forward lookup requests for unknown constants to the slaves as the partitions are normal RDF-3X files which are able to provide any piece of metadata about the triples contained there.

4.2.2 The Slaves

The slaves are lightweight server processes running at different machines and listening for execution requests coming either from the coordinator or from other slaves. They are always associated to a partition from the original database and are bound to a host name and a port. A partition is a standard RDF-3X file with a single peculiarity: it uses a shared string-to-integer mapping among the other partitions, rather than building a new one as the standard RDF-3X tool for data import would do. The execution requests consist of serialized parts of a query plan, which the slave deserializes and executes. Slaves do not waste time on local query optimization because the coordinator has all necessary statistics to optimize the query completely. Slaves send results to their requestors in pages of 1024 values.

4.3 Global Query Optimization

The global query optimizer uses the information stored in the metastore to generate a cheap plan according to a cost function. Traditionally, cost models for query execution are expressed with respect to either the **total time** or the **response time**. The **total time** is the sum of all possible time components, even if they are supposed to run in parallel. Therefore, it is a measure to quantify resource consumption. On the other hand, the response time measures the latency of answering a query with two existing variants: (a) time until the first result are output and (b) time until the last result is output [45]. Although a centralized cost model for RDF data might optimize for response time (e.g. the RDF-3X cost model), it is not directly

applicable to a distributed environment because it does not consider communication costs, which have a significant impact in such environments.

PARTOUT's global query optimization algorithm avoids exhaustive search as determining costs for all possible plans would be too expensive in a distributed environment. Therefore, it applies a two-steps approach that starts with a plan that is optimized with respect to cardinalities and selectivities from the whole triple store and applies heuristics to make the plan more efficient in the distributed environment. As the RDF-3X optimizer is known to produce efficient query execution plans, we use such a plan as the start point for our optimization process. RDF-3X cost model defines the cost of a query plan in terms of the response time until the last result is produced. In the remainder of this chapter, we will first discuss the RDF-3X optimizer and highlight the differences with our system. After having introduced PARTOUT's distributed cost model, we will show how these components work together to find an optimized query execution plan for the distributed setup.

4.3.1 Initial Query Plan

PARTOUT builds upon and extends query plans used in RDF-3X. This system stores triples in relations with all possible permutations of S,P, and O (SPO, SOP, PSO, POS, OSP, OPS) which are called *Facts Indexes*. Additionally, RDF-3X maintains indexes with aggregated counts for all possible permutations of two attributes (SP, SO, PS, PO, OS, OP) and one attribute (S, P, O) which we call the *Aggregated* and *Fully Aggregated Indexes*. RDF-3X' query optimizer uses a bottom-up dynamic programming algorithm which finds an optimal solution for a big problem, i.e., the whole query, based on the optimal solutions of its subproblems, i.e, parts of the query. It first seeds a table with the index scans for the base relations that correspond to the query triple patterns. The type of index depends on whether all the variables will be needed afterwards. For example, if a variable in a triple pattern is not used anywhere else, the optimizer will choose Aggregated or Fully Aggregated indexes. Note that projections conceptually preserve the cardinality because aggregated indexes store the counts for the combinations of values. Starting with the seeds, a set of larger plans is created by joining optimal solutions of smaller problems. Those larger plans correspond, for example, to all possible orders to join a set of triple patterns. Then the RDF-3X cost model uses the statistics contained in the aggregated indexes and selectivity histograms to estimate the response time of those plans and prune the ones with high estimated response time. It is important to mention that in PARTOUT, those statistics do not come from a standard RDF-3X file but from the metastore built at partitioning time. Figure 9(a) shows an example query and the

corresponding start plan obtained by the RDF-3X optimizer using the statistics in the metastore.

URIs and string literals are mapped to unique integer ids which are used in all indexes. Leaf-level operators in query plans are therefore index scans corresponding to triple patterns in the query. The triple pattern *(?s,rdf:type,db:city)* results in a scan of the *POS* facts index, retrieving matching subjects in increasing id order. This is exploited for join operators which are implemented by cheap merge joins whenever the two inputs are ordered by the join attribute; if that is not the case, a hash join is applied.

4.3.2 Distributed Cost Model

For reasons of compatibility to the initial plan, PARTOUT reuses the RDF-3X cost model and extends it with the aspect of communication in distributed systems.

Physical plans in RDF-3X use pipelining whenever possible. The only exception to this rule are pipeline breakers, such as the sort operator, that need to read the complete input before producing the first result.

In distributed systems, it would be too expensive in terms of the number of messages to send each tuple immediately to another host. Thus, to minimize communication costs, PARTOUT batches data transfers between hosts in pages of 1024 results. The benefit of batching results is especially significant for non-selective queries because in these cases communication costs are the most important factor in the response time.

To compare different execution plans, we need a cost function $c(plan)$ that describes the execution costs for plan $plan$. The cost $c(plan)$ of a plan is determined by the cost $c(RootOp)$ of its root operator:

$$c(plan) = c(RootOp) \tag{9}$$

The cost of an operator op consists of its execution cost $xc(op)$ (the estimated response time to evaluate it, which can be estimated by the RDF-3X optimizer), the costs of its child operators (because of exploiting parallelism in combination with response time, we only need to consider the maximum of all children), and possibly the cost $tc(x, op)$ to transfer the results from the child operators (potentially located in other hosts):

$$c(op) = xc(op) + \max_{x \in children(op)} (tc(x, op) + c(x)) \tag{10}$$

We can estimate the cost to transfer results from operator x to operator op by

$$tc(x, op) = \begin{cases} t_{page} \cdot \left\lceil \frac{excard(x)}{1024} \right\rceil & ; hh(x) \neq hh(op) \\ 0 & ; \text{otherwise} \end{cases} \quad (11)$$

Here, $hh(op)$ denotes the home host of an operator and $excard(op)$ is the expected output cardinality of operator op as estimated by the RDF-3X optimizer. We assume that communication costs for one page of data are symmetric, constant, and equal to t_{page}, which has to be empirically measured for a specific setup. Still, t_{page} can be easily replaced by more complex models and therefore adapted to environments with heterogeneous network topologies and distances between hosts.

4.3.3 Query Planning

The first stage of PARTOUT's two-steps query planning is to generate a start plan based on the statistics for the entire dataset provided by the metastore. For this purpose, PARTOUT uses the RDF-3X optimizer [44]. Leaf nodes represent index scans that correspond to triple patterns contained in the original query. In contrast to centralized systems, the triples matching these graph patterns are spread across the hosts in the distributed setup. Thus, we need to determine the hosts that are relevant to the leaf node scans. We call this process, the multiple sources resolution phase.

A host may provide results for a triple pattern if the pattern overlaps with at least one minterm of a fragment allocated to that host according to the metastore. Figure 9(b) shows an example query with its start plan, where each leaf operator is annotated with the hosts that may contain relevant triples for it. 9(c) shows the pieces of information in the relation PredicateHost of the metastore used to determine that the predicates implied by the index scan overlap with a fragment allocated at some host. If there is only a single relevant host for a leaf operator, the scan is exclusively executed at that host. If there is more than one relevant host, the operator is executed at each host and their results are combined through a chain of binary merge-union (BMU) operators as illustrated in Figure 9(d). A BMU operator takes two sorted inputs and merges them preserving sort order.

On the other hand, index scans for triple patterns whose implied predicates are not known by the metastore do always hit the remainder fragment which according to the policy for the avoidance of imbalance discussed in Chapter 3.5, overlap with all hosts since their triples were uniformly distributed among all the partitions.

After the multiple sources resolution phase, the optimizer will apply a set of trans-

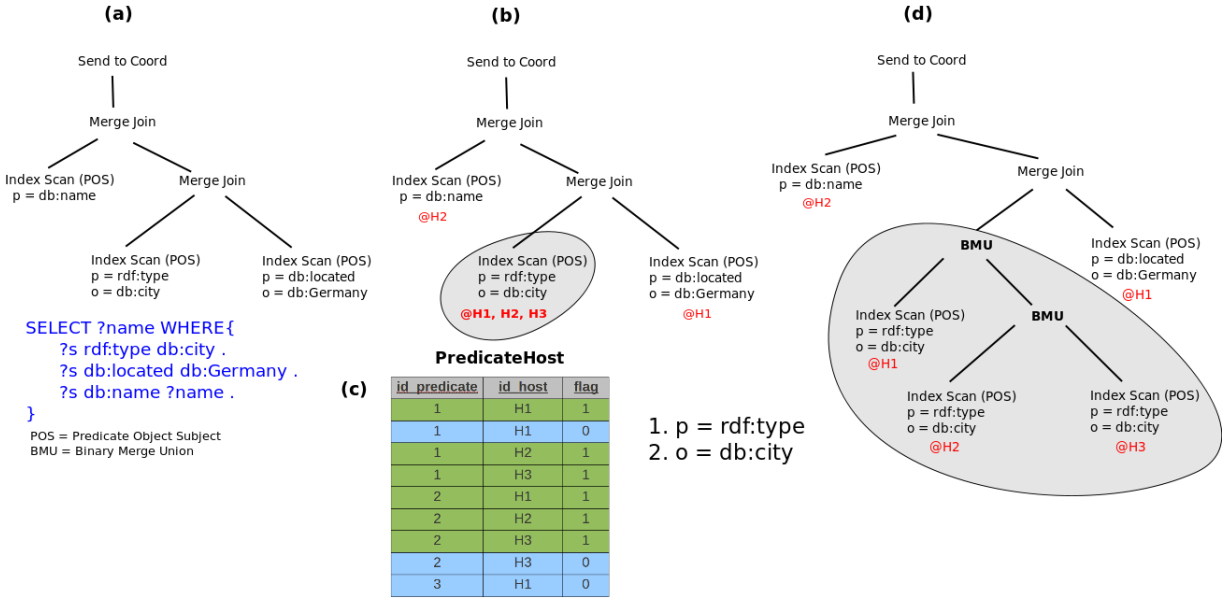


Figure 9: (a) A start execution plan. (b) The same plan annotated with relevant hosts for leaf operators. (c) The mappings in the metastore that determine the relevant hosts in this example (d) Resulting execution plan after replacing scans on multiple hosts with chains of binary merge union operators.

formations to the obtained plan in order to produce a perhaps new and cheaper plan in terms of response time. For this purpose, it defines two types of transformations, DJ and AHH.

The $DJ(rootOp)$ (for Distribute Join) transformation intends to exploit parallelism by distributing joins in the following manner: (i) identifying joins in the hierarchy rooted at $rootOp$ with leaf scans as inputs and at least one of them relevant to several hosts (ii) adding a MergeJoin operator for each combination of index scans in the left and right sides of the join and (iii) combining partial results with binary merge union (BMU) operators. Especially for non-selective queries with many intermediate results, DJ can significantly reduce response time because many tuples can be processed in parallel. Figure 10 (b) shows the effect of the DJ transformation on our example query. It shows how the input subtrees of a Merge Join operator are distributed in order to increase parallelization.

The second transformation named $AHH(RootOp, host)$ (for *Assign home host*), sets *host* as home host of *RootOp* which means the operator will be evaluated at that place. The argument *host* is the identifier of a host in the cluster. AHH is normally applied to the root operator in the query plan in order to assign an evaluation place to the non-leaf operators in the tree. Its pseudocode is described in Algorithm 2. It works recursively, by first traversing the tree downwards until the leaves have been

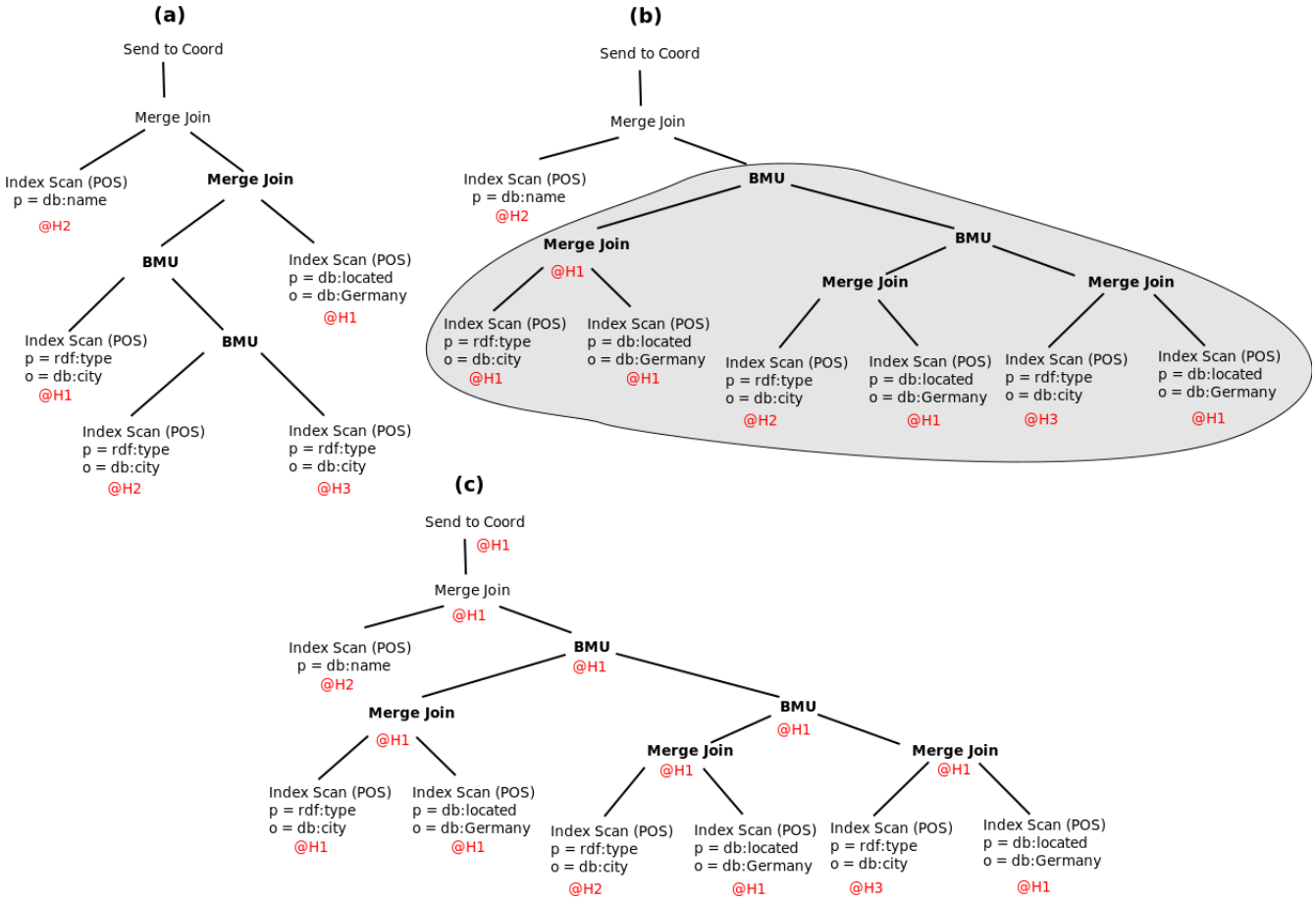


Figure 10: (a) Example query plan (b) The result query plan after the application of the DJ (c) The result after the application of $AHH(RootOp, h_1)$.

reached and collecting their home hosts which were assigned during the multiple sources resolution phase. Lines 3-8 illustrate this first step. Then the algorithm goes up and tries to assign the argument *host* as the home host to each inner node in the operator tree, respecting this heuristic: An operator can be evaluated only at one of the home hosts of its children. If the argument *host* is not in the set of home hosts of the children, then the host incurring in fewer data transmissions is picked. This is achieved by selecting the host of the operator with the biggest expected cardinality according to the RDF-3X cost model (lines 16 and 17). This heuristic has however an exception. BMU operators are allowed to be evaluated in a host which is not in the set of home hosts of their children; as a general rule, all the BMU operators of a chain are evaluated at the same host, which is the host of the parent operator (e.g a join) of the top BMU in the chain (lines 19-24). This exception is aimed to avoid unnecessary transmissions as all the data relevant to an index scan is transmitted once and collected in parallel at the same place where it will be processed. The

AHH returns *true* if at least one of the leaves had already the argument *host* as its home host. Figure 10(c) shows the effect of applying AHH with host h_1 as argument to the query plan in 10(b).

```
 1: relevantHosts ← {}
 2: bmus ← {}
 3: if rootOp.isLeaf() then
 4:    return rootOp.homeHost = host
 5: end if
 6: for childOp ∈ rootOp.children do
 7:    outcome ← outcome ∨ AHH(childOp, host)
 8:    relevantHosts = relevantHosts ∪ {childOp.homeHost}
 9:    if childOp.type = BMU then
10:       bmus ← bmus ∪ {childOp}
11:    end if
12: end for
13: if host ∉ relevantHosts then
14:    outcome = outcome ∨ false
15:    if bmus = ∅ then
16:       highestCardOp ← findHighestCardOperator(rootOp.children)
17:       rootOp.homeHost ← highestCardOp.homeHost
18:    else
19:       if |bmus| = 1 then
20:          rootOp.homeHost ← bmus[0].homeHost
21:       else
22:          highestCardOp ← findHighestCardOperator(bmus)
23:          rootOp.homeHost ← highestCardOp.homeHost
24:       end if
25:    end if
26: else
27:    rootOp.homeHost ← host
28:    return true
29: end if
30: return outcome
```

Algorithm 2: $AHH(rootOp, host)$ tries to assign *host* as the evaluation place for the query plan rooted at *rootOp*.

The query planning routine in Algorithm 3 proceeds in a greedy manner. It starts with a query plan whose leaves have already a home host. At each step, it applies a transformation (AHH or DJ) to obtain a new plan. If the transformation improves the response time according to the cost model, the algorithm takes it. If k is the number of relevant hosts for the all the leaves of the plan, the algorithm will try AHH k times, one per relevant host. Then if DJ is feasible, it is applied and if it improves the response time, a second round of k AHH transformations is scheduled.

```
relevantHosts ← startPlan.getRelevantHosts()
for host ∈ relevantHosts do
  transformations ∪ {AHH(host)}
end for
transformations ← {}
if startPlan.appliesForDJ() then
  transformations ∪ {DJ(rootOp)}
  for host ∈ relevantHosts do
    transformations ∪ {AHH(rootOp, host)}
  end for
end if
currentPlan ← startPlan
for transform ∈ transformations do
  newPlan ← applyTransformation(currentPlan, transform)
  if cost(newPlan) < cost(currentPlan) then
    currentPlan ← newPlan
  else
    if transform = DJ then
      break
    end if
  end if
end for
return currentPlan
```

Algorithm 3: SearchCheapPlan(startPlan) applies a set of transformations to startPlan in order to find a new query plan with the lowest cost.

4.4 Query Execution

Query execution in PARTOUT consists of two phases. The preparation phase where the obtained query plan is recursively transmitted from the coordinator to the slaves, and the execution phase where the data is again recursively retrieved and shipped to the coordinator.

The coordinator starts the preparation phase by serializing the obtained query plan and sending it to the home host assigned to the root operator. This home host, in turn, deserializes the received message and converts it into an executable physical plan. If one of the received operators is assigned to another home host, the subtree rooted by this operator is replaced with a *Remote Fetcher* operator that will later receive results for the subtree. The subtree itself is serialized and sent to the indicated home host, which instantiates a *Remote Sender* operator at the root of the received query plan that will send all obtained results for the subtree to requestor host. This procedure is repeated whenever the received subplan contains operators assigned to other hosts. Every recipient will recursively report success

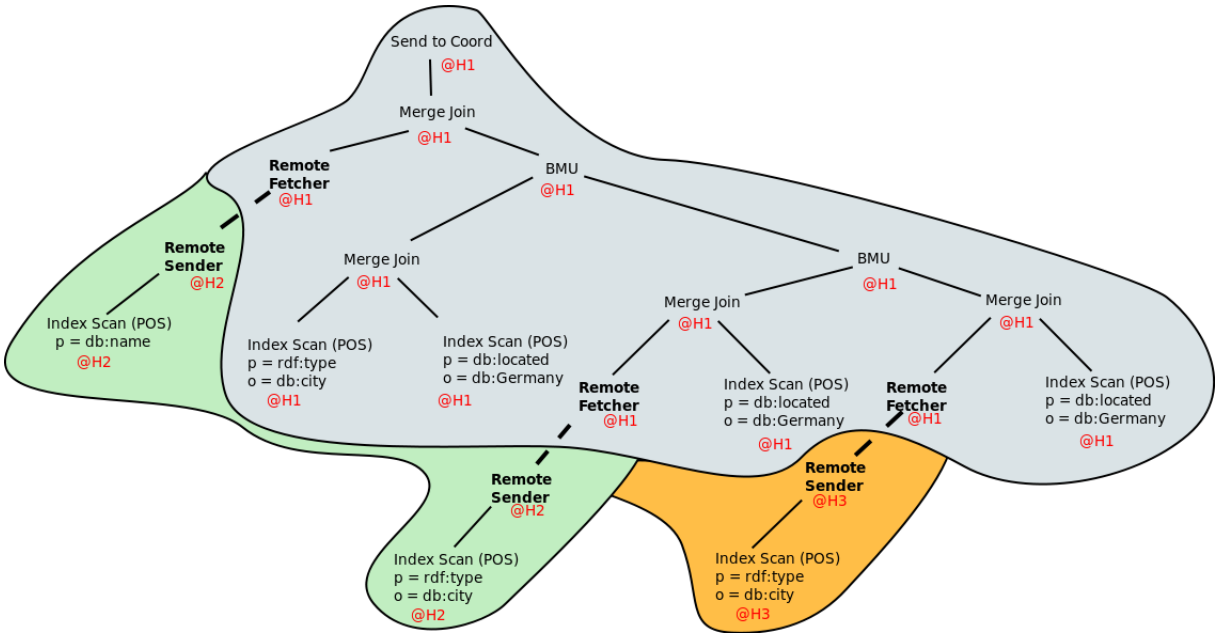


Figure 11: Example plan after solving the remote edges through pairs of sender/fetcher operators.

to its requestor (including the coordinator who initiated the process) as soon the operation is completed.

Once all the operators in the query plan have reached their home hosts, the coordinator sends an execution request to the home host of the top operator which recursively propagates the request to all the slaves hosting subtrees of the query plan. The results for the root nodes of such subtrees are forwarded to other hosts according to the corresponding *Result Sender* operator when either one page of results is filled or all results have been generated. At the end, the coordinator has received all the results for the query. Figure 11 illustrates the query execution plan that we obtain for our example query.

5 Implementation

PARTOUT consists of three software components which are built on top of the RDF-3X engine. They are a query load aware partitioner and allocator, a query coordinator and a slave server. The two latter correspond to the coordinator and slave components already introduced in Chapter 4. This chapter is intended to describe the implementation of those components, the most relevant challenges addressed during their development and how they interact with the RDF-3X engine implementation.

5.1 The RDF-3X distribution

PARTOUT's development was based on the version 0.36 of RDF-3X available at `http://code.google.com/p/rdf3x`. An RDF-3X triple store is a single binary file containing the data, indexes and statistics on top of a fully native and efficient storage implementation. An RDF-3X file is divided into internal *database partitions* which we call *regions* to avoid confusion with our previous definition of partition. Each region contains a set of *segments*, the building blocks of an RDF-3X database. Every index in the triple store is associated to a segment which is internally implemented as B+ tree divided in data pages and provides a simple programming interface through a public class and a set of methods. This includes the Facts, Aggregated and Fully Aggregated indexes (accessible through the classes FactsSegment, AggregatedFactsSegment and FullyAggregatedFactsSegment), the selectivity histograms (StatisticsSegment class) and the strings dictionary (DictionarySegment class) which is internally compounded of two subcomponents, for string-to-integer and integer-to-string lookups. All indexes use the unique integer identifiers assigned to every string at the database construction and stored in the strings dictionary.

From the user point of view, the RDF-3X distribution provides a set of programs for the most important tasks of RDF data management. We can mention:

- **rdf3xload**: It is a simple utility that takes one or more files containing a set of RDF triples in the N3 format (Notation 3) [23] and generates a RDF-3X triple store file.
- **rdf3xquery**: This program takes a RDF-3X triple store and a text file containing a query and outputs the results. Additionally, it provides a very simple command line interface to execute queries in a interactive way. There also exist a feature to print the query execution plan without running it.

[23] `http://www.w3.org/TeamSubmission/n3/`

- **rdf3xdump**: It does the opposite to *rdf3xload*. Given a RDF-3X triple store file, it outputs all the triples in N3 format.

The distribution includes many other tools targeted for developers; however the three aforementioned utilities provide the core functionality of a database management system and are relevant to the development of PARTOUT. All the programs in the RDF-3X distribution rely on a common set of classes or API which provides features for SPARQL and N3 parsing, query planning, query execution, data storage and retrieval (the segments classes) and basic concurrency control. Our system uses thoses classes intensively. RDF-3X runs in Windows and GNU/Linux for 32 and 64 bits, however it is strongly recommended by the author to use it on 64 bits platforms for big datasets.

5.2 Query Load Aware Partitioner and Allocator

The utility *qloadpartitioner* is a program which takes a RDF-3X triple store, a query load and the number of hosts in the cluster n as input and outputs the metastore for the coordinator [24] and n disjoint partitions, each one associated to a different slave. The metastore is a relational SQLite database [25], whereas the partitions are regular RDF-3X triple store files generated with a tweaked version of the *rdf3xload* utility. The query load can be provided as a set of files containing dbpedia access logs or plain SPARQL SELECT queries. Since query loads for running systems can be huge, *qloadpartitioner* samples it randomly using *Reservoir Sampling* [64], a technique designed to sample data streams whose size is unknown or very big. It is worth mentioning that before accepting a query in the sample set, the partitioning utility verifies if it is representative, which means the query could potentially have results because all its constants have entries in the strings dictionary. *qloadpartitioner* uses the RDF-3X classes for parsing SPARQL to build the query graph and then invokes the static analysis feature that lookups constants and detects unsatisfiable queries in advance.

The next step extracts simple predicates from the triple patterns and filter conditions of the surviving queries (now transformed into query graphs) and use them to feed an implementation of the optimal horizontal fragmentation method described in Chapter 3. This algorithm relies on statistics about the sizes of generated minterms which are obtained through an extension of StatisticsSegment

[24] See Chapter 4.2.1 for more details
[25] `http://www.sqlite.org/`

class which provides a set of methods to estimate the cardinalities for triple patterns and joins which in behind uses classes FactsSegments, AggregatedFactsSegment and FullyAggregatedFactsSegment. In particular there is a method to get cardinality of an arbitrary combination of constants (which define a triple pattern): $getCardinality(subject, predicate, object)$. For example, the size of the fragment represented by the minterm $property = const_1$ is obtained through a call to $getCardinality(0xFFFF, string2integer(const_1), 0xFFFF)$ which internally does a lookup in the P Fully Aggregated Index by means of the FullyAggregatedFactsSegment class, to obtain the exact count. $const_1$ is replaced by its integer identifier and the unbounded positions (subject and object) are denoted using the integer $0xFFFF$.

For simple predicates containing operators besides Equal (=), PARTOUT extends the $getCardinality$ method to accept also extra simple predicates. The list of supported predicates consists of $isIRI(a)$, $langMatches(a, const)$, $datatype(a) = uri$ and $a\ op\ const$ (with $a \in \{subj, prop, obj\}$, $op \in \{<, \leq, =, \geq, >\}$, $literal \in L$, $const \in (U \cup L)$ and $uri \in U$). This method estimates the cardinality of a minterm by evaluating the predicates over a sample of the triples contained in the facts indexes. Consider a modification of our example where now the minterm is $property = const_1 \wedge langMatches(object, \text{“}de\text{“})$. Moreover, suppose the previous call to $getCardinality$ returns k. Depending on how big is k, the method will evaluate the second predicate against the k triples matching $property = const_1$ or just against a subset of them. The evaluation of the predicates requires a cheap lookup in one of the facts segments (PSO or POS in this example) for the data page with triples such that $property = const_1$ and then either a full iteration or a sampling of the region to evaluate $langMatches(object, \text{“}de\text{“})$. When k is big, which is normal for minterms like $property = rdf : type \wedge langMatches(object, \text{“}de\text{“})$ with low selective properties or simply $langMatches(object, \text{“}de\text{“})$, where we need a sample over the whole triple store, the method assigns a random integer-id value to one of the unbounded variables producing a pivot location. The method $find(subject, property, object)$ from the FactsSegment class looks for the successor of the pivot: the smallest existing triple which is greater than the pivot location. Once a successor is found, it is evaluated against the predicate. This process is repeated $log_2(k)$ times providing an estimation for the selectivity of the predicate based on how many triples in the sample evaluated to true. If there are h of such triples then the selectivity is estimated as $\frac{h}{log_2(k)}$ and the cardinality of the complete minterm is $\frac{h}{log_2(k)} \times k$. Moreover, RDF-3X provides a way to know the biggest integer used as a dictionary id, which reduces the probability of producing pivot locations without a successor triple.

The next big step of the *qloadpartitioner* is to allocate the fragments to the hosts based on the method described in Chapter 3.4. Once the allocation is done, it is time to build the actual partition files, which implies to evaluate the fragments definitions against every single triple in order to determine the host it belongs to. Host summaries can reduce the number of predicate evaluations substantially by identifying decisive predicates in a host in such a way that the triples do not have to evaluated against every minterm assigned to the host. Nevertheless, the number of triples in big datasets still represents a challenge in terms of time, therefore the implementation exploits parallelization by assigning regions of the facts segment to every processor. Given the fact that patterns with bounded properties are normally the least selective, it builds a shared queue with all possible properties (easily obtained from the P Fully Aggregated Segment) which is consumed by different threads. Once a thread gets a property value, it will evaluate the fragment definitions against all the triples matching this triple pattern, translate them to their string (N3) representation and output them to the temporary file of the right host. When it has finished with one region, it dequeues another property value and repeats the process. It is important to remark that the access to the temporary files must be also synchronized since they can be written concurrently by all threads. In order to reduce contention, writes are batched so that the locks on the temporal files are not acquired in a per-triple basis. Furthermore, this phase is also in charge of gathering the integer-to-string mappings which will be needed by the query coordinator at processing time. They are stored in a shared temporal file too. The compile time argument $FULL\_DICT$ allows to control whether the whole dictionary is kept in the metastore or just a fraction whose size is determined by the argument (also compile time) $DICT\_MAX\_SIZE$. If just a fraction of the dictionary is stored, missed dictionary lookups have to be forwarded to the slaves.

The next stage of the partitions building phase requires to convert the temporal N3 files produced in the previous stage into actual RDF-3X triple stores. Unfortunately, the *rdf3xload* utility is not useful for this purpose because it creates a new id space for every partition. To force a global id space shared by the coordinator and the slaves, a modified version of *rdf3xload* has been written. This program takes an additional argument: the string-to-integer mappings source which in this case is the original RDF-3X file, and assigns to every string the integer identifier it had in the original triple store.

The final step of the query load aware partitioner populates the tables in the metastore with the information required for query processing: the dictionary entries, the predicates definitions and the statistics for triple patterns and joins, which are obtained from the newly built partitions.

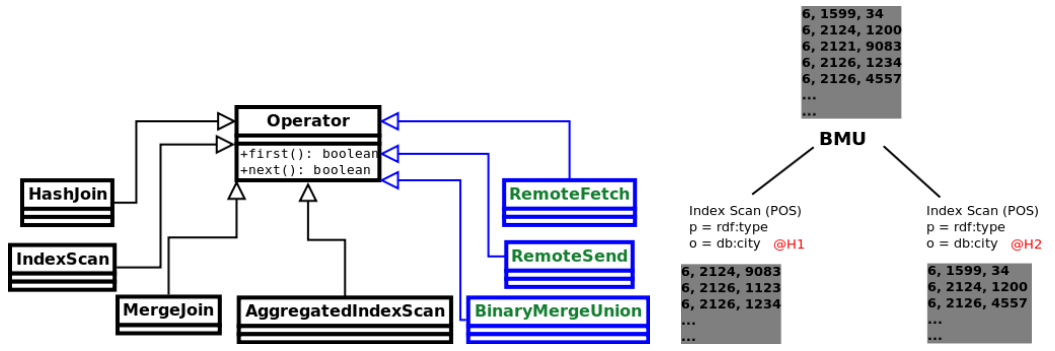
5.3 The Query Coordinator

The query coordinator *partoutquery* is a simple program which resembles the utility *rdf3xquery* with a key difference: it requires a PARTOUT's metastore instead of a RDF-3X triple store file as input. Except from that difference, it provides the same features though it contains some structural changes. The most relevant are:

- Rewrite of the query planner using the same logic but retrieving the statistical information from the metastore rather than from the Statistics class.
- Adaptation of the resulting query plan to run in a distributed setup through the implementation of the BMU (Binary Merge Union) physical operator.
- Extension of the RDF-3X cost model to consider network transfers
- Serialization of the query plans for transmission to the slaves.

Query execution in RDF-3X has four well-defined stages: query parsing, static analysis, query planning and query execution. From those stages, only the first one remains identical, whereas the fourth was partially reused. Query parsing produces a structured representation of the query and its components which is forwarded to the static analysis module to produce a query graph with triple patterns as nodes and edges for every potential joining variable. The query planning phase takes a query graph and produces a query plan, a tree data structure describing the hierarchy of operations (index scans, joins, selections) needed to answer the query. The last stage takes a query plan and produces a physical plan which can be executed over the data. The implementation of the static analysis and query planning phases includes method signatures which use objects tailored for a RDF-3X triple store file like the dictionary and index segments classes, therefore they were rewritten using our new classes with identical interface but different implementation. For example, the RDF-3X class DictionarySegment has a counterpart DDictionarySegment (D for distributed) which provides the same methods as the original class but instead of performing a lookup in the RDF-3X B+ trees, it issues an SQL query against the Dictionary table of the metastore and in case of a miss, forwards the lookup to the slaves and caches the answer for future use. Utility data structures like the QueryGraph and Plan data

types (used solely for data representation) are effectively reused, but the conversion from the query plan to the physical plan implemented in the RDF-3X class CodeGen was extended to adapt the resulting plan for a distributed setup. Physical plans in RDF-3X are trees whose nodes are instances of different classes, all of them subtypes of the abstract class *Operator*. This class provides a simple interface for pipelined execution as Figure 12(a) shows, therefore an operator will forward data to its parent as soon as it has results. The method $first()$ is invoked to retrieve the first set of bindings for an operator whereas the subsequent bindings are obtained by calling $next()$ until it returns $false$. In a normal execution, the invokation of $next()$ is propagated down the operator tree hierarchy to achieve a pipelined execution. The BMU operator is intensively used when the bindings for an index scan operator are distributed among several hosts. As index scans produce sorted results, BMU implements a standard merge algorithm to produce a unique fully sorted stream of triples, giving the impression that the data is read from a local non-fragmented index as Figure 12(b) shows. Right after the multiples sources has been solved by means of the BMU operator, the query coordinator uses the aforementioned AHH and DJ transformations to find an optimal evaluation place for the non-leaf operators, in terms of the new cost model. PARTOUT's cost model introduced in Chapter 4.3.2 reuses the RDF-3X cost model for response time and extends it to consider the cost of evaluating the new BMU operator as well as the cost of network transmissions.



(a) The RDF-3X Operator hierarchy

(b) The BMU (Binary Merge Union) operator

Figure 12: (a) The Operator class and some of its subclasses. The highlighted subclasses were added to support distributed execution. (b) The BMU operator takes two sorted sets of bindings and produces a fully sorted set using a standard merge algorithm.

The query execution starts with the transmission of the physical plan, from the query coordinator to the slave server running at the host where the root operator

was assigned for evaluation. PARTOUT relies on the library Protocol Buffers[26] for data serialization over the network. This library provides an extensible, efficient and platform-neutral way of serializing structured data, in this case all the possible messages between the coordinator and the slaves which include:

- Dictionary lookups: integer-to-string (*Int2StrDictLookup*) and string-to-integer (*Str2IntDictLookup*)
- Prepare for execution (*Prep4Exec*)
- Execute (*Exec*)

The static analysis phase which converts a parsed query into a query graph, requires to convert the strings contained in the query into RDF-3X ids. When the full dictionary is not kept at the metastore and in the presence of lookup misses, the DDictionarySegment class broadcasts *Str2IntDictLookup* messages to all the slaves. Lookups are batched for sets of keys to reduce latency and bandwith utilization. When a slave receives a *Str2IntDictLookup* request for a set of strings, it will lookup them in its local dictionary and provide the integer values for the ones it knows.

The *Prepare for Execution* message, shortened as *prep4Exec* is sent from the query coordinator to a slave or from one slave to another and includes a serialized query plan. This message is aimed for preparing the recipient for the execution of the provided operator tree. For the sake of brevity, Figure 13(a) shows the example query used in previous sections and its query plan without all the transformations applied during query optimization. Its serialization using the Debug utility of Protocol Buffers is depicted in 13(b). The physical plan is serialized as an array of operators obtained by traversing the operator tree in a breath search manner. In general, the transmitted information for an operator includes the type, the expected cardinality based on the estimations of the cost model and the relative memory locations for the data bindings (denoted with the prefix register). They tell the operators where to find the input and write the output. The recipient slave is now in charge of distributing the portions of the plan evaluated at other places, to the other slaves by means of more *Prep4Exec* messages. This process is recursively performed until all the operators have been successfully transfered to their home hosts. If a slave was able to transfer all the subparts of the plan he received to the corresponding hosts, it will reply success to its requestor. The response message includes a unique request

[26] `http://code.google.com/p/protobuf/`

id which identifies the execution request.

The last stage of the query execution consists of sending *Exec* messages to the target slave until there are not more results to retrieve. The *Exec* requests must provide the unique id the server assigned to the request in the preparation phase. *Exec* responses include the actual result bindings in batches of 1024 integers and a boolean value stating that there might be more results. As soon as the query coordinator gets a data batch, it translates the received integers into strings (by using the class DDictionarySegment) and presents them to the user. This process is repeated until the more-results flag in the response becomes false.

Finally it is important to mention that the introduced two-stages execution was designed to benefit the system's availability in certain scenarios and at some extent. If during the preparation phase, one slave is not available due to a crash or network partition, the query execution could in some cases continue at the risk of providing incomplete results and the query coordinator can know it in advance. That is particularly true when the failing slave hosts only a portion of an index scan; however there can be more harmful cases like the unavailability of the top slave or some other slave in charge of a join evaluation. In those cases it is probably better to report the query as unable to run.

5.4 The Slave Server

The slave server *partoutslave* is a lightweight process that is in charge of a data partition and process requests involving that data. The requests includes the messages described in the previous section. The utility takes a host name, a TCP port number and a RDF-3X triple store and relies on the library Boost.Asio [27] for network operations. It mainly consists of an instance of the class boost::asio::io_service which provides the functionalities of an asynchronous server that processes requests using a pool of threads whose size can be also provided as an optional argument. By default, it will use as many threads as available processors.

The slave server deserializes the requests and executes them accordingly. *Int2StrDictLookup* and *Str2IntDictLookup* requests are processed as dictionary lookups against the partition's dictionary, whereas *Prep4Exec* messages require the assignment of a unique id (within the slave server) to the request as well as the the deserialization of the transmitted operator tree to turn it into an executable RDF-3X

[27] `http://www.boost.org/doc/libs/1_48_0/doc/html/boost_asio.html`

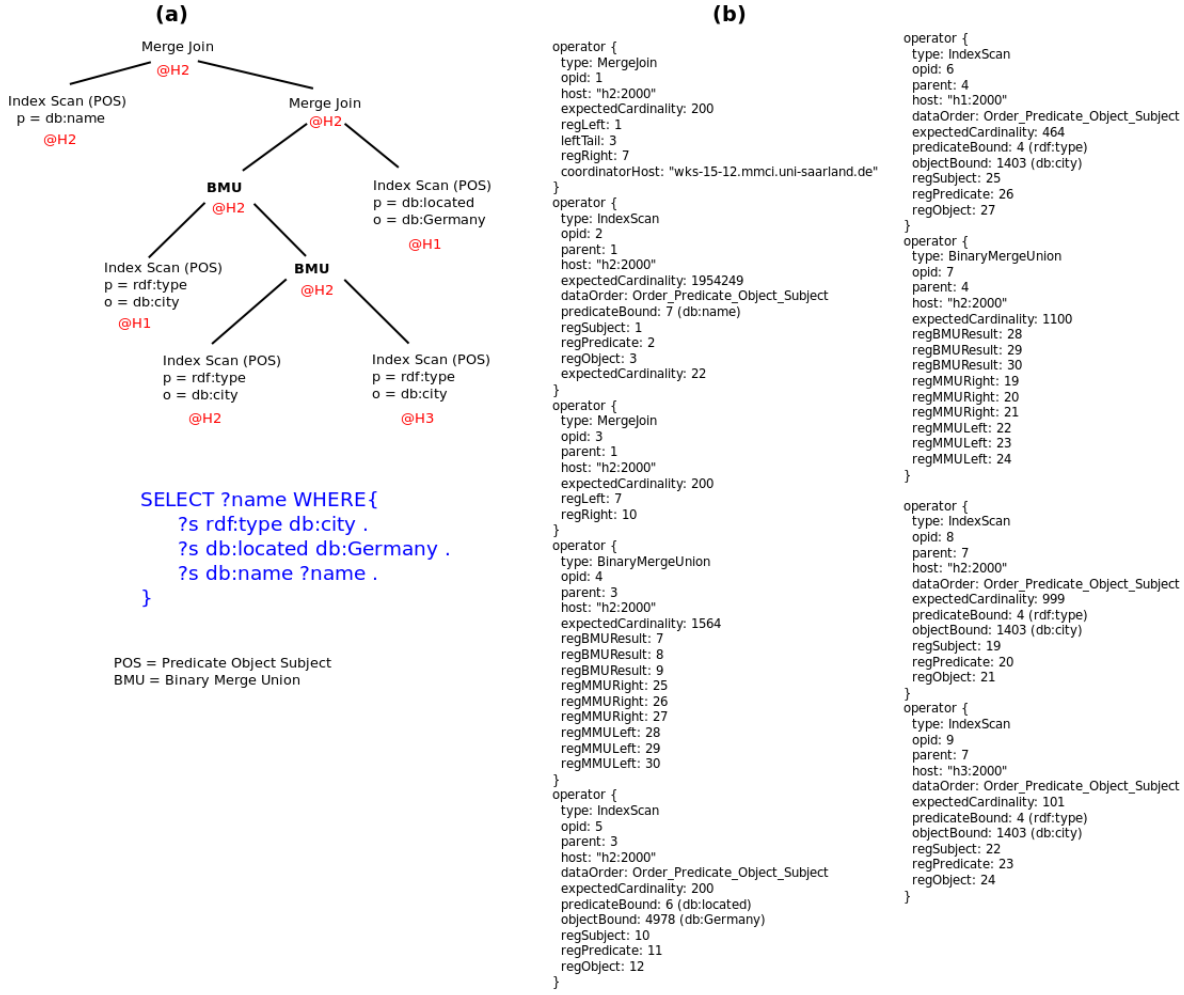


Figure 13: (a) A example execution plan. (b) The same plan serialized with Protocol Buffers

physical plan. The parsing routine scans the serialized operator tree (which responds to a breadth search traverse) and converts the serialized operators into actual physical operators while their home host is the host where *partoutslave* is running. As soon as it finds a subtree rooted at an operator whose home host is different, it (a) produces a *Prep4Exec* message for that subtree and sends it to the remote host and (b) adds a RemoteFetch operator which encapsulates the remote read, creating the illusion that there is a single centralized plan. At the end, it adds a RemoteSend operator at the top of the operator tree to encapsulate the data shipping to the requestor. When the process is finished, the slave server replies with a successful *Prep4Exec* response. Note that a single slave could potentially receive more than one *Prep4Exec* request associated to disconnected portions of the original plan like in Figure 14. This situation exploits parallelization since every subtree will be evaluated in a different

thread even though some operators might be transmitted to the same host more than once.

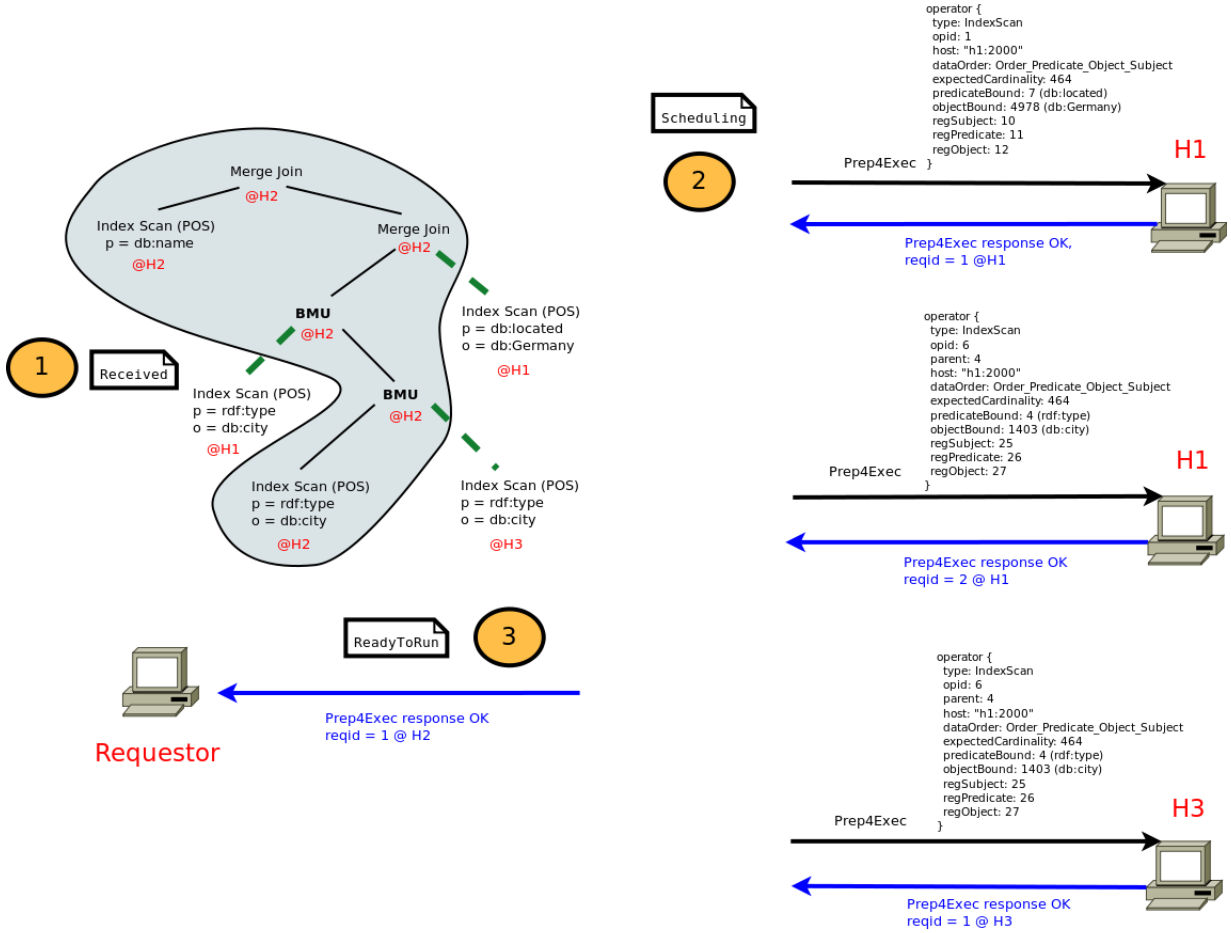


Figure 14: The flow of processing a request at the slave server. (1) The slave server deserializes the portions of the plan to be evaluated locally. (2) It prepares the other slaves by sending the subplans they are in charge of. (3) It replies to the requestor with an id for the request used later on to start the execution.

When *partoutslave* receives an *Exec* request, it first checks the id of the *Prep4Exec* message it refers to. Once it has found it, it starts the pipelined execution of the operator tree by calling the method $first()$ of the root operator which is propagated down the hierarchy in order to retrieve the first set of bindings. Subsequent bindings are obtained through calls to the $next()$ method. The RemoteSend and RemoteFetch operators aimed for encapsulating data shipping and reception inherit the Operator interface, however RemoteFetch is not truly pipelined because it actually sends an *Exec* request to fetch a whole batch of bindings when $first()$ is invoked, then buffers them and finally retrieves the first one. Subsequent calls to $next()$ simply consume the bindings buffer until it is empty. If that happens and the last *Exec* response

stated there might be more results, the client fetches the next batch. In order to improve response time and avoid waiting for the whole page to be transmitted, the RemoteFetch class actually requests the next batch of bindings in background while the last one is processed by the upper operators. This process is repeated until the *Exec* reply states that all bindings were transferred.

On the other hand, the RemoteSend operator is simply a wrapper which retrieves all the bindings of its child operator, serializes them and writes in the socket.

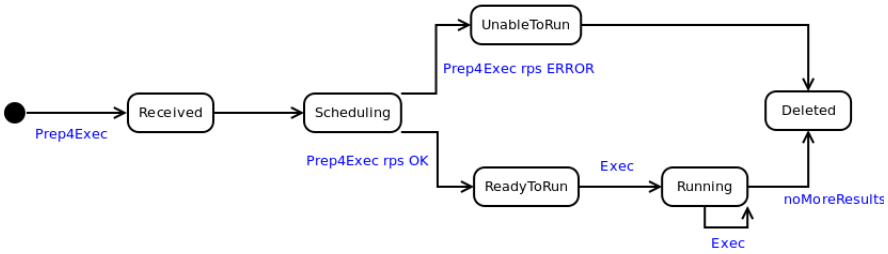


Figure 15: State diagram for the life cycle of a execution request.

Finally Figure 15 depicts the state diagram for the life cycle of an execution request. When the *Prep4Exec* message arrives at the server, the request starts with the state Received. When the server has deserialized the region of the tree supposed to be evaluated locally, it changes to Scheduling. This state denotes that the server is preparing other slaves by sending them the subplans they were assigned to. If this operation reaches successful outcome, then the request moves to the ReadyToRun state, otherwise it becomes UnableToRun. At this point the slave server replies to the requestor waiting for the first *Exec* message. When it arrives, the request goes to the state Running until all bindings have been retrieved and sent. At that point, the request can be deleted. Note that after a request has reached the *ReadyToRun* state, there is a hazard for memory overusage if the *Exec* never arrives. The same can happen during the *Running* state if the physical plan has not retrieved all the data but the requestor has not asked for another batch of bindings. Even though the current version of *partoutslave* does not implement this feature, a simple background garbage collector thread could solve this problem. In the first case, the request arrival timestamp could be used. If a request has been in the *ReadyToRun* state longer than a given threshold, it is simply removed from the requests map. For the second case, a modification timestamp is required in order to register the last time the operator tree was asked to report results.

6 Evaluation

The first round of experiments is aimed to compare the performance of query processing using the fragmentation and allocation implemented in PARTOUT against a centralized execution and a naive partitioning method on four datasets, three of them taken from SPARQL benchmarks. This performance is defined in terms of the total response time; the time elapsed between the initiation of the query and the retrieval of the last result. A second round of short experiments was run in different setups in order to measure the impact of network transmissions in the overall response time in PARTOUT. The chapter starts with a brief description of the testing data (triple stores and query load) and the different setups. Then it describes the conditions under which the experiments were run, to provide a proper explanation of the obtained results. It finally concludes with a brief analysis of the impact of remote communication and the number of partitions in the overall response time.

6.1 Datasets

Three of the datasets used for the experiments were obtained from well-known benchmarks in the field of RDF and SPARQL. Sp2bench is one of the datasets of the Fedbench benchmark [28] and includes 10M RDF triples of synthethic bibliographic information. Berlin20K and Berlin500K are datasets produced by the data generator tool included in the distribution of the Berlin SPARQL Benchmark [29], using a scale factor of 20K and 500K, including approximately 7M and 180M triples of semantic data about an e-commerce use case where a set of products is offered by different vendors and consumers have posted reviews about the products. They both have the same schema (e.g. same properties and resource types) but differ in the number of resources that are described. On the other hand, *Billion triple challenge* is the third and biggest dataset and was taken from the edition 2008 of the Semantic Web Challenge [30]. It contains more than 500M triples of semantic data taken from DBpedia and other sources which are part of the Linking Open Data cloud. Table 6 summarizes the most relevant information about the testing datasets.

The number of different properties in a dataset can be considered as a measure for the complexity or diversity of the data schema. For the *Billion triple challenge* dataset, the high number is not surprising since this dataset is a compendium of

[28] `http://code.google.com/p/fbench/`
[29] `http://www4.wiwiss.fu-berlin.de/bizer/berlinsparqlbenchmark/`
[30] `http://challenge.semanticweb.org/`

several interlinked datasets. Note also that it poses a challenge when finding a proper experimental query load that tries to bind as much data as possible.

Table 6: Summarized information about the testing datasets

Dataset	*RDF-3X file size (GB)*	*# triples*	*Properties dataset*	*Properties query load*
Sp2bench	0.871	10M	76	14
Berlin20k	0.775	6.95M	31	40
Berlin500k	19	175.41M	31	40
Billion triple challenge	54	562.50M	79977	24

6.2 Queries

The first round of experiments used 20 different queries taken both from the benchmarks and also from crawling the data graph. Ten of them were used as input for the *qloadpartitioner* utility with the same frequency of occurrence (once). Regarding the structure of the queries, they were carefully chosen to provide diversity in terms of three criteria: topology (star-shaped, paths and combinations of them), number of triple patterns (between 1 and 10), and result selectivity in order to produce a varied set of execution query plans. For example, pure star-shaped queries tend to produce chains of Merge Join operators whereas path queries cannot exploit the sorted structure of the indexes forcing the query planner to use Hash Joins. Data projections in other respects have an effect on the type of indexes used by the query planner. In general, aggregation functions like *count*() or triple patterns with an unbound position whose variable is not used anywhere else will make the query planner to use the Aggregated or Fully Aggregated indexes which incur in less data to be transmitted. Table 7 provides a brief description of the testing query set; the detailed list of queries can be found in Appendix A.1.

Table 7: Summary about the testing and training queries

	Triple patterns			Shape			Cardinality		
	1-3	4-7	>7	Star	Path	Mixed [31]	<1K	1K-10K	>10K
Sp2bench	13	4	3	7	3	10	12	2	6
Berlin20K	9	11	0	14	1	5	9	1	10
Berlin500K	11	9	0	5	1	14	6	6	8
Billion triple challenge	4	12	4	6	14	0	17	2	1

6.3 Opponents

Two characteristics of PARTOUT are assessed in this chapter: the quality of the fragmentation and allocation plus its impact on query processing and the general system performance. Since the goal of PARTOUT's cost model and query processor is to minimize response time, the system performance is measured in terms of this metric. Moreover, the quality of the fragmentation and allocation scheme and its impact on query processing is evaluated by comparing our proposed method against a naive approach like fragmentation by property using the same query planner: PARTOUT's query processor. On the other hand, the general system performance, aimed to show the feasibility of our method for scalable RDF management, is assessed through a comparison against a fully centralized and efficient execution, in this case a single RDF-3X file containing the whole dataset.

Fragmentation by property was effectively achieved by providing the right input to the utility *qloadpartitioner*, i.e., simple queries with a single triple pattern only with bound property like `SELECT * WHERE {?s rdf:type ?o}`. The list of properties was limited only to those occurring in the query load used to test PARTOUT. This decision is based on the fact that some existing RDF solutions like Jena [9] or Sesame [7] provide the possibility to fragment the data internally by property, allowing to cluster triples with important properties in dedicated relations while keeping the rest in another relation which resembles our remainder fragment. This can help reduce file and index sizes for properties with low selectivity and speed up data lookup. Often, it is also combined with an approach that maps properties onto a property table [66] which can be a materialized view with one row per resource and one column per property so that a tuple actually encodes the information of several RDF triples describing one resource. This approach is especially beneficial for star-shaped queries which try to find a set of resources and in general good when triple patterns containing those property values join frequently in the query load. The fragmentation by property achieved by PARTOUT clusters properties in a partition without considering any join information, but only an "assign where it fits" policy. Since the input query load does not contain joining triple patterns, in every step of the allocation routine, the benefit of assigning a fragment to a host is determined exclusively on the number of triples the host already has, hence the fragment is assigned to the host with fewer triples allocated at that moment. Additionally, the remainder fragment containing the triples with properties not mentioned in the query load is splitted among all hosts as described in Section 3.5. Finally, the meta-store contains only information about simple predicates of type `property=const` (with `const` occuring in query load), the hosts relevant to them and their cardinalities.

6.4 Setup

All the experiments were implemented on top of the version 0.3.6 of RDF-3X available at `http://code.google.com/p/rdf3x/`. The first round compares the response time of PARTOUT in the aforementioned datasets, using 5 and 10 partitions in a distributed setup against the opponents described in the previous section, whereas the second round uses the *Billion triple challenge* dataset with 10 partitions and a short list of very simple queries in order to analyze the impact of distribution and remote transmissions in the overall performance. For this purpose, different setups were used. They are described in Table 8.

Table 8: Different setups used in the experiments

Setup	Query coordinator	Slave servers
1	1 machine with an Intel Xeon E5530 processor @2.40GHz, 32 GB of RAM running Debian GNU/Linux 5.0.9 for 64 bits	5 machines with an Intel Xeon E5430 processor @2.66GHz, 32 GB of RAM running Debian GNU/Linux 6.0.2 for 64 bits
2	1 machine with an Intel Xeon E5530 processor @2.40GHz, 32 GB of RAM running Debian GNU/Linux 5.0.9 for 64 bits	The same machine as the query coordinator
3	1 machine with a Intel Core(TM) i5 CPU 760 @2.80GHz, 8GB of RAM running Debian GNU/Linux 5.0.9 for 64 bits	5 machines with the same characteristics as the query coordinator

Note that besides Setup 2, the experiments using 10 partitions required to run 2 slave servers per host. Setup 2 runs both the query coordinator and the slaves servers in the same machine. Setups 1 and 3 use a different machine for the query coordinator, so no slave server was running at the same host as the coordinator.

6.5 Experiments and Results

The first round of experiments compares the total response time in query processing for 20 queries when using PARTOUT's partitioning utility with only 10 of them as input, fragmentation by property and the centralized execution in a single RDF-3X file using Setup 1. Testing the system with queries not used for partitioning is aimed

to show the impact of changing the query load on system's performance. For the centralized case, the utility *rdf3xquery* was slightly modified in order to collect the relevant timing information. Moreover, *rdf3xquery* was tested in one of the slave machines of Setup 1. As the response times for the different queries can vary from a few milliseconds to several minutes, all the charts use logarithmic scale for readability.

6.5.1 Sp2bench

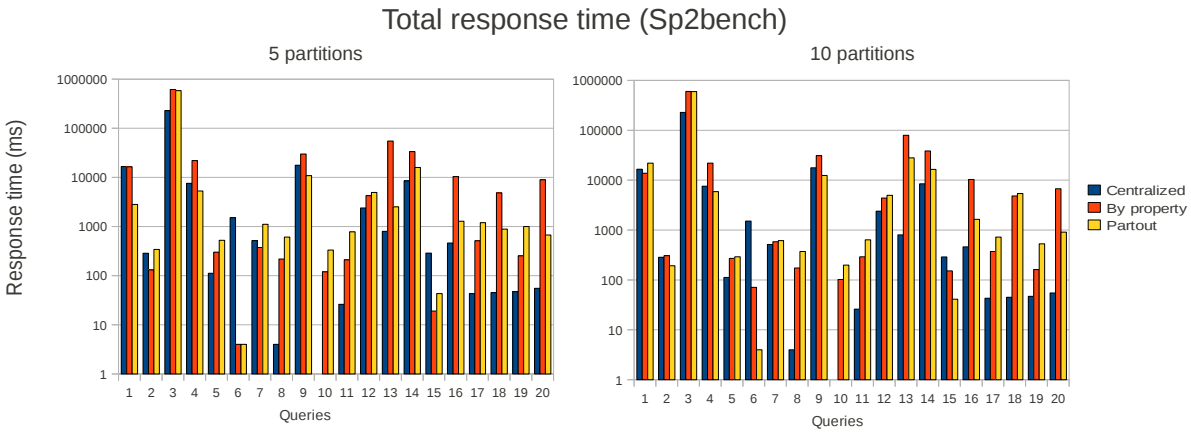


Figure 16: Response time for the *Sp2bench* dataset

The results for the *Sp2bench* dataset are depicted in Figure 16. The first ten queries were used for training. As discussed in the previous chapter, communication costs normally dominate the response time of distributed executions. Nevertheless, we see that for queries 1, 4, 6, 9 and 15 (5 partitions), PARTOUT produces lower response times than any other approach. A look at the query plans for such queries reveals a small cost in terms of data transfers because most of the bindings for the triple patterns are distributed across only one host in some cases. For example, query 6 has a single triple pattern which hits always a single partition. Query 1 hits one and two partitions for the 5 and 10 hosts cases respectively, in spite of consisting of 10 triple patterns. Likewise, query 9 consisting of 8 triple patterns hits always only two partitions but unlike the others, it has low selectivity incurring in a longer delay when shipping the results to the coordinator. The reason a query hitting a single host runs faster than the centralized approach is that it is executed over a subset of the whole triple store, much smaller in size, resulting in a faster local execution. Moreover, the highly selective query 15 deserves some special attention as it runs faster with 5 partitions with PARTOUT. The rationale behind the results for this query are

explained by two facts: (a) the query hits all hosts in both cases which means the query plan for 10 partitions is bigger and (b) the amount of intermediate results is small, thus the response time is not fully dominated by communication costs. When considered together, both conditions increase the impact of local execution in the total response time.

In most cases, fragmentation by property results in higher response times since the assignment of bindings for the triple patterns is somehow random, resulting in more relevant hosts and therefore bigger chains of BMU operators. Because the queries used to feed the partitioner do not include join occurrences, the allocation phase will simply take every fragment and assign it to the host with more available space at that moment. For query processing, it means that there is no guarantee that two joining triple patterns have their bindings in the same host which sometimes incurs in high communication costs. Note however that fragmentation by property achieves better times for queries 2 (with 5 partitions), 5 and 8, all of them extremely selective, with few triple patterns and bounds in the subject and object positions. In all these cases, it could be observed that the query plans obtained with PARTOUT contained chains of BMU operators because their bindings were split accross 2 or 3 fragments which were assigned to different hosts by the allocator. This makes sense if we remember that the allocation routine can assign a fragment to an empty host when the hosts containing joining fragments are overloaded. Despite the parallel fetch of the data, BMU chains introduce some delay at execution time as can be seen for these queries. The plans produced with fragmentation by property hit several partitions but every triple pattern was relevant only to one partition not requiring to merge data at all.

Finally, notice that in general and more particularly for untrained queries, the distributed execution imposes a penalty up to two orders of magnitude in the response time in comparison to the centralized execution. This is particularly evident for queries 8, 10 and 17-20.

6.5.2 Berlin500K and Berlin20K

The datasets *Berlin500k* and *Berlin20k* were both tested to illustrate the impact of scaling the amount of data in query processing. Even though they share the same schema i.e., same properties, resource types and relationships, they are not tested with exactly the same query load as the query load for Berlin20K was not fully representative to Berlin500K. Nonetheless, both sets of training queries follow the guidelines previously described in Section 6.2.

The query loads for these datasets have the best coverage in terms of used

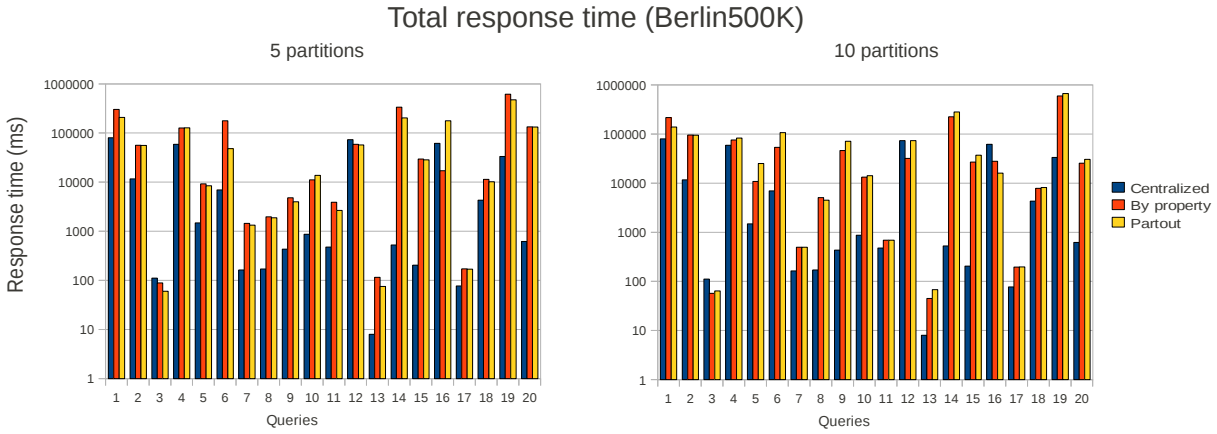


Figure 17: Response time for the *Berlin500K* dataset

properties; 31 out of 40 are used to train the partitioner. Their results are shown in Figures 17 and 18. As Figure 17 shows, PARTOUT beats the centralized execution in queries 3 and 12 with *Berlin500K*, which are opposite in characteristics: the first one very simple (1 triple pattern) and highly selective, the second consisting of 4 triple patterns and more than 250K results. In general the cost of distribution is more evident in this dataset as the response time of both the distributed approaches tends to increase, producing delays up to 3 orders of magnitude in comparison to the centralized execution for queries like 14, 15 and 20. The situation is the same in the smaller counterpart *Berlin20k* where the centralized execution always produces lower response times than the distributed approaches according to Figure 18. Note however that the gap between the centralized approach and PARTOUT gets smaller as the data grows since the majority of the queries in *Berlin20K* run two orders of magnitude slower in the distributed setups.

Moreover, our approach exhibits higher response times than fragmentation by property for the trained query 10 and the untrained query 16 in *Berlin500K* because of the same reasons as presented for query 15 in *sp2bench*: local execution delay introduced by long chains of BMU operators with significant influence on the response time. However this scenario has a particularity: query 16 improves its performance when using 10 partitions which might look counterintuitive with our previous statements. This query hits all the partitions in both cases, but unlike query 15 in *Sp2bench*, the amount of intermediate results is huge. This means the impact of longer chains of BMU operators is completely outweighed by the communication costs, which now are smaller for the 10 partitions case because the transmitted

amount of data per remote link is smaller and shipped in parallel. This scenario is an example of the benefits of parallelism.

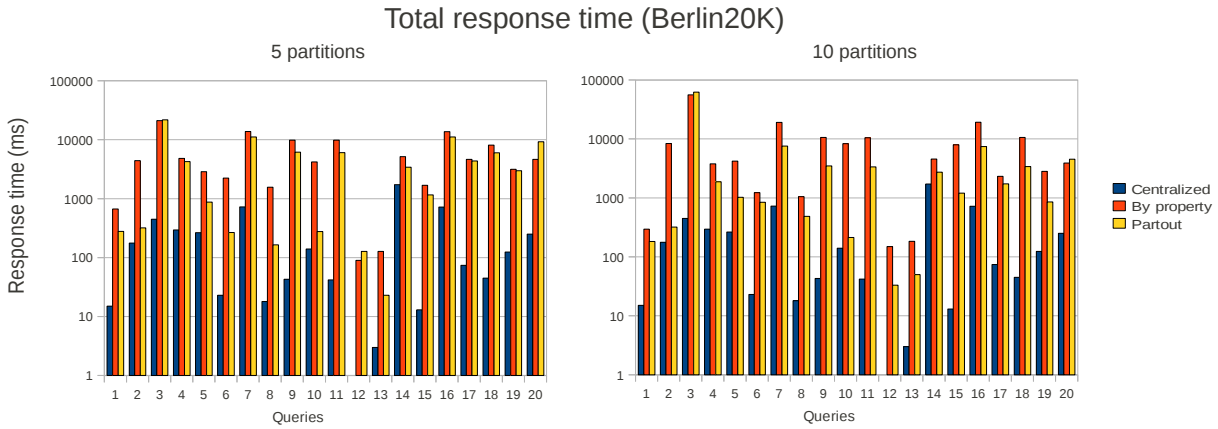


Figure 18: Response time for the *Berlin20k* dataset

6.5.3 Billion triple challenge

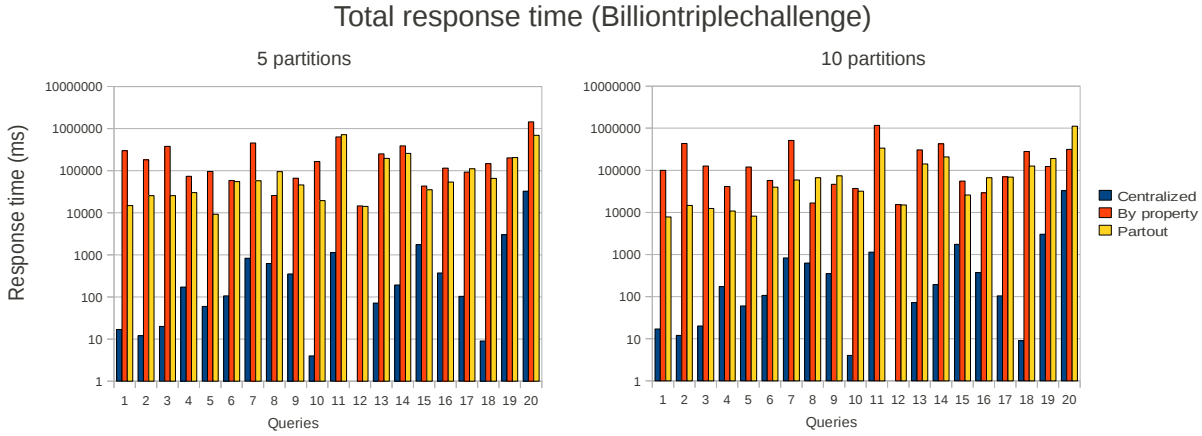


Figure 19: Response time for the *Billion triple challenge* dataset

The results for this dataset are depicted in Figure 19. They show a high penalty due to communication costs, resulting in response times up to 3 orders of magnitude bigger than the centralized execution, but considerably better than fragmentation by property in most of the cases. Unlike the previous databases, the RDF schema is pretty diverse which combined with a low-coverage query load (24 out of nearly

80K properties) results in a very skewed distribution for the load (size multiplied by its data access) of the fragments as Figure 20 shows. The remainder fragment was not considered in this analysis, because it is not assigned to any host, but evenly distributed among the cluster. We can observe that 90% of the total load is concentrated in less than 10% of the fragments. It is important to mention that the load is mostly influenced by the size of the fragments since each query appears once in the query load and a triple pattern rarely occurs in more than three queries. For the allocation, it implies those big fragments will be likely located alone in a single partition splitting the bindings for trained queries among several hosts incurring always in communication costs even though the query planner will transmit the smallest of two separated joining fragments during query execution.

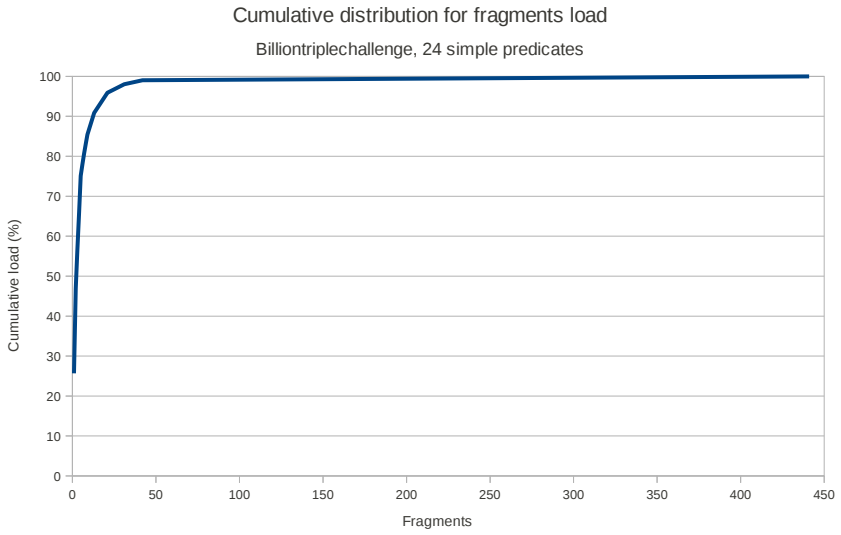


Figure 20: Cumulative load for the fragments of the *Billion triple challenge* dataset

6.5.4 The Impact of Distribution

It is a well known fact in Distributed Systems that communication costs are the most striking factor in the response time [15, 45] because distribution always introduces an additional level of complexity. In transactional systems with strong consistency requirements for example, it requires the implementation of complex distributed consensus protocols with several rounds of messages. In other respects, latency in systems running on WANs (e.g. the Internet) is dominated by network latency. The goal of this section is to provide an insight of how splitting the data among different hosts actually affects the response time in PARTOUT, in comparison to a fully centralized approach where all the data resides in the same disk.

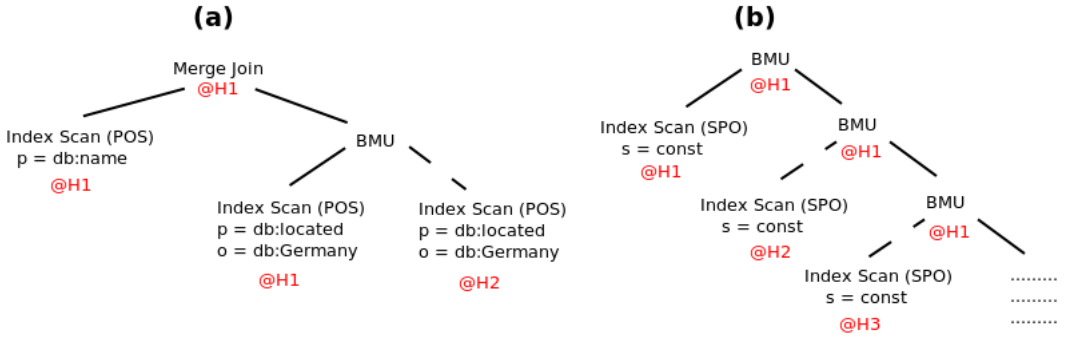


Figure 21: (a) A query plan distributed at 2 hosts. (b) A single triple pattern relevant to all hosts.

Even though it is always possible to estimate the time spent in network transmissions, quantifying their impact in the overall response time in the presence of parallelism is not obvious. Consider the example in Figure 21(a) which shows a very simple query plan: a merge join of two index scans, one of them with bindings in two hosts. Even though the time to ship the results of the join to the requestor can be easily measured, the impact of distribution in the response time of the join operator is not clear. To see why this is true, consider the case when the first bunch of results comes from joining the values produced by the local index scans. That could happen if the values of the local scan on the right side of the join are sorted before the values of the remote scan. It means the fetched data is not actually used inmediately, but it will be local when required because the BMU must transfer at least one page from the remote scan, to report the first tuple to the join operator. In this case the only possible impact of remote communication might come from the fetch of the first page. Additionally, consider what happens if the left scan in the join introduces a high latency because the local machine is busy so that it responds after the remote data on the right side was fetched. In that case, the operator response time is dominated by the local execution and the characteristics of the data. In general, it is hard to measure the fraction of an operator's response time induced by remote transmissions or local waits. In spite of this uncertainty, it is still possible to estimate the impact of communication by bounding this value. The second round of experiments was intended to define a lower bound for the impact of communication in the total response time for a particular scenario. Four simple queries with one single triple pattern and only one bound constant were used to measure this impact in the *Billion triple challenge* dataset with 10 partitions.

- `SELECT * WHERE {`$const_1$ `?p ?o}`
- `SELECT * WHERE {?s` $const_2$ `?o}`

- `SELECT * WHERE {?s ?p` $const_3$`}`
- `SELECT * WHERE {?s ?p` $const_4$`}`

All of them except the last were designed to hit the remainder fragment so that they have bindings in all hosts, thus their plans consist of a single BMU chain with 10 leaves like depicted in Figure 21(b). The fourth query used an object value with low selectivity occurring in the query load, with bindings in 7 hosts. We define c_c as the estimated communication cost and r_c as the communication impact ratio for this scenario.

$$c_c := t_{prep} + t_{merge} + t_{ship} \tag{12}$$

$$t_{merge} := t_{response}(topBMU) - max\{t_{response}(is_1), \ldots, t_{response}(is_n)\} \tag{13}$$

$$t_{ship} := t_{response} - t_{plan} - t_{prep} - t_{response}(topBMU) \tag{14}$$

$$r_c := \frac{c_c}{t_{response}} \tag{15}$$

c_c estimates the amount of actual time that is induced by distribution whereas r_c expresses this time as a fraction of $t_{response}$. We denote by t_{prep} the time to distribute the query plan to all involved slaves by means of the *Prep4Exec* message. t_{merge} is the execution cost of merging the data in the BMU chain and is calculated as the total response time of the top BMU operator minus the maximum response time among the involved index scans. The total response time of an operator, denoted as $t_{response}(op)$ is the time elapsed between the retrieval of the first and last set of results. Note that in a centralized setup, there would not be any need to merge data, therefore we count this time as part of the communication cost. t_{ship} measures the time spent exclusively in transmissions between the coordinator and the home host of the top BMU and is calculated substracting the planning and preparation phase times and the execution time of the top operator from the total response time. We claim that the formula for c_c provides just a lower bound for the actual cost of remote communication as it does not consider the delays of any operator due to remote waits. This is because it is impossible to know with such a simple model, which part of the total response time of an index scan for instance, is caused by local or remote waits. Table 9 shows the results for $t_{response}$, c_c and r_c for our queries tested in setups 1, 2 and 3.

Table 9: Values for the response time and the bounds for the estimated communication cost shown milliseconds and in percentage of the response time

	Times (ms):	$t_{response}$	t_{prep}	t_{merge}	t_{ship}	c_c	r_c in %
Setup 1	Query 1	48	14	1	23	38	79.17
	Query 2	6362	16	1	115	132	2.07
	Query 3	63	11	1	35	47	74.60
	Query 4	205715	34	0	37	71	0.03
Setup 2	Query 1	48	14	4	15	33	68.75
	Query 2	7539	16	0	17	33	0.44
	Query 3	28	13	1	13	27	96.43
	Query 4	241059	11	2	13	26	0.01
Setup 3	Query 1	48	12	2	13	27	56.25
	Query 2	37243	8	1	10	19	0.05
	Query 3	45	11	1	12	19	21
	Query 4	170323	6	2	6	14	0.01

The cases shown in Table 9 provide just a hint about the impact of communication in PARTOUT because they describe very simple scenarios. As stated in Table 8, setups 1 and 3 are distributed with five hosts running two instances of the slave servers and a different host for the coordinator. They use machines with different characteristics of CPU and memory. On the other hand, setup 2 runs all the slaves in the same physical host which effectively reduces the values for c_c because there are no network transmissions. Note however that the values for $t_{response}$ are higher in this setup because the queries are very simple and therefore strongly influenced by CPU time, which is slightly greater because the processors have a lower clock rate than the ones in setups 1 and 3. As queries become more complex, the dimensions of the query plans as well as the delay of the preparation phase (which affects t_{prep}) become bigger because deeper query plan trees with more operators are recursively distributed among the slaves. Moreover, the situation gets worse with more hosts especially if a leaf operator has bindings everywhere. Finally, we can observe that the impact of distribution is much more obvious for simple highly selective queries, where most of the response time is induced by the communication costs.

6.5.5 Increasing the Number of Hosts

The current evidence provided by our rounds of experiments does not suggest any corelation between the response time and the number of hosts as Figure 22 shows. In general, the effect of having more partitions and hosts in a cluster can be benefitial or harmful depending on the characteristics of the query. As shown in the experiments'

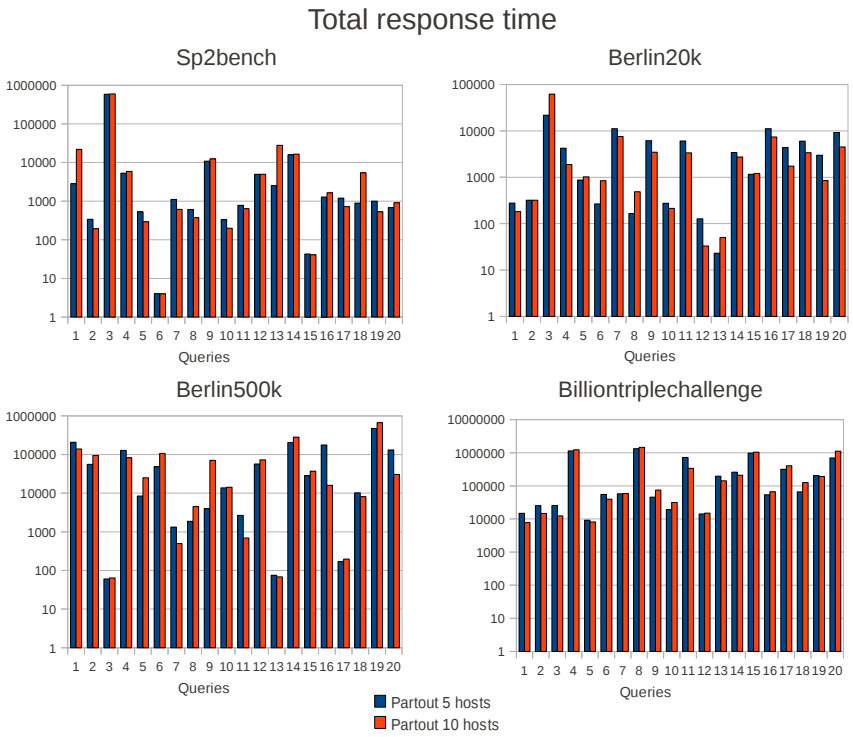


Figure 22: Response time for the different datasets with 5 and 10 partitions

discussion, the effect of a bigger cluster in combination with properly trained queries is benefitial most of the times, since more hosts imply smaller partitions and therefore faster local execution. However, due to the load balancing policy considered in the allocation routine, having more partitions increases the chances of splitting the bindings for complex queries among several hosts, increasing communication costs but also offering more opportunities for parallelization. Queries with many intermediate results especially benefit from high parallelization, whereas for queries with little intermediate data, the complexity of the query plan and its impact on local execution becomes significant. It is important to remark, however, that these results should not be taken as a discouragement to increase the number of machines in the cluster as they might benefit other system properties like throughput. More partitions imply that queries might have bindings splitted among more hosts but also that the bindings relevant to unrelated queries will be probably located in different machines, balancing the load properly and exploiting parallelization so that a single host is not in charge of the whole data processing in the presence of many concurrent requests.

7 Conclusion

In this work, we have proposed a solution for scalable processing of RDF data in a cluster of machines by considering the two major aspects of its design: data fragmentation and allocation, and distributed query processing. The experimental results as well as the previous analysis, reveal its feasibility for certain cases, including small and medium-size data sets, where we achieve better response times than a centralized execution in a very efficient triple store like RDF-3x. However, the experiments also suggest many opportunities for improvement.

Regarding the fragmentation and allocation stage, this work provides a method to extract access patterns from a SPARQL query load and use them to partition a RDF dataset using standard methods for relational data. It is important to remark that our approach does not consider replication at all, even though it could be easily extended to support it. The key of this assessment is the allocation strategy. In every iteration, the allocation algorithm assigns a fragment to the host with the highest benefit at that moment, where the benefit is a function of the host's current load and the number of join operations that would become local if the fragment were assigned to the host. If a fragment is relevant to more than one query with relevant fragments allocated in different hosts, the benefit for such hosts will be raised, but only one can have the fragment, imposing a penalty to some queries. A straightforward solution to avoid this penalty could assign the fragment to all hosts containing joining fragments or, in the presence of storage space constraints, to the k most benefitial hosts where k could be fixed or calculated in every iteration based on the distribution of the benefits. Moreover, the aforementioned scenario has a close relation with the discussion about the impact of increasing the number of hosts, as more partitions increase the probability of this scenario to arise.

In other respects, the structure of the query load represents another source of improvement; in particular the information encoded in the global fragment query graph. For example, a query load containing groups of queries with disjoint sets of triple patterns produces a global fragment query graph with disconnected components. This could be exploited by the allocation strategy, by allocating such disconnected components to disjoint subsets of the cluster to make sure the bindings for the queries in a group are not splitted among many hosts. This idea suggests further analysis to find a relation between the complexity of the query load and the minimal number of hosts that are enough to guarantee fully local operations. If there are more hosts in the cluster than the minimal, then replicate the data in the other hosts. Furthermore, approaches based on graph partitioning like Schism [15] are appealing for the global fragment query graph because of its moderate size.

In relation to distributed query processing, PARTOUT proposes a two-stages query optimization process which takes a start query and makes it suitable for distributed execution. Even though the current implementation is built on top of RDF-3X, the essence of the method is platform-agnostic. Additionally, depending on the underlying software, there is a lot of room for improvement. For example, since the partitioning routine extracts predicates not only from triple patterns but also from filter conditions and uses them to build fragments, the metastore could provide not only good estimations for the cardinality of selections like $object > 10^9$ (not available in RDF-3X) but also accurate information about which hosts are actually relevant to a filter condition over a triple pattern so that selection operators are only sent to the relevant partitions. The current version of PARTOUT does not offer this possibility but it could be achieved by extending the RDF-3X query planner so that it considers the cost of pushing selections up or down in the query plan.

Additionally our system model could be used for other tasks besides query processing, specifically update management and repartitioning. The design of PARTOUT relies on some assumptions about RDF triple stores, specifically the way updates normally occur. As discussed in Chapter 2.2.1, in RDF stores, updates are rare and tend to append triples instead of modifying the existing ones. Even though updates management has not been considered in our analysis, it could leverage the current architecture. If a new triple arrives at the coordinator, it can be easily forwarded to the right partition by simply finding the minterm for which it evaluates to *true* and then the host in charge of the minterm.

Finally, a fragmentation scheme can become suboptimal for a triple store if either the data or its access patterns change. In the presence of updates, the size component of the load can lead to imbalance as some partitions get more load than others, whereas changes in the query load affect the frequency of access. In such cases, repartitioning is required and could easily use the current architecture by splitting the partitioning work among all hosts. The fragmentation and allocation routines would be executed at the coordinator and the results forwarded to the slaves, in charge of sending the triples to the new hosts they belong to.

References

[1] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009.

[2] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. Van Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In *ISWC*, pages 107–121. 2004.

[3] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD*, pages 359–370, 2004.

[4] C. B. Aranda, M. Arenas, and Ó. Corcho. Semantics and optimization of the SPARQL 1.1 federation extension. In *ESWC (2)*, pages 1–15, 2011.

[5] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1st edition, 1997.

[6] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[7] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. In *Spinning the Semantic Web*, pages 197–222, 2003.

[8] M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW*, pages 650–657, 2004.

[9] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the Semantic Web recommendations. In *WWW (Alternate Track Papers & Posters)*, pages 74–83, 2004.

[10] P. Castagna, A. Seaborne, and C. Dollin. A parallel processing framework for rdf design and issues. Technical Report HPL-2009-346, Hewlett-Packard Labs, 2009.

[11] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *SIGMOD*, pages 128–136, 1982.

[12] S. Chakravarthy, J. Muthuraj, R. Varadarajan, and S. B. Navathe. An objective function for vertically partitioning relations in distributed databases and its analysis. *Distributed and Parallel Databases*, 2(2):183–207, 1994.

[13] H. Choi, J. Son, Y. Cho, M. K. Sung, and Y. D. Chung. SPIDER: a system for scalable, parallel / distributed evaluation of large-scale RDF data. In *CIKM*, pages 2087–2088, 2009.

[14] D. W. Cornell and P. S. Yu. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Trans. Software Eng.*, 16(2):248–258, 1990.

[15] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[16] P. Dadam. *Verteilte Datenbanken und Client/Server-Systeme*. Springer, 1996.

[17] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[18] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35:85–98, June 1992.

[19] A. Dimovski, G. Velinov, and D. Sahpaski. Horizontal partitioning by predicate abstraction and its application to data warehouse design. In *ADBIS*, pages 164–175, 2010.

[20] A. Doan, L. Gravano, R. Ramakrishnan, and S. Vaithyanathan, editors. *Special issue on information extraction. SIGMOD Record, 37(4)*, 2008.

[21] R. Domenig and K. R. Dittrich. An overview and classification of mediated query systems. *SIGMOD Record*, 28(3):63–72, 1999.

[22] O. Erling. Implementing a SPARQL compliant RDF triple store using a SQL-ORDBMS. http://virtuoso.openlinksw.com/whitepapers/SPARQLL-ORDBMS.pdf.

[23] I. Filali, F. Bongiovanni, F. Huet, and F. Baude. A survey of structured P2P systems for RDF data storage and retrieval. *T. Large-Scale Data- and Knowledge-Centered Systems*, 3:20–55, 2011.

[24] G. H. L. Fletcher and P. W. Beck. Scalable indexing of RDF graphs for efficient join processing. In *CIKM*, pages 1513–1516, 2009.

[25] Franz Inc. AllegroGraph. http://www.franz.com/agraph/allegrograph/.

[26] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. In *SIGMOD*, pages 93–101, 1979.

[27] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *WWW*, pages 411–420, 2010.

[28] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the Web. In *ISWC/ASWC*, pages 211–224, 2007.

[29] O. Hartig. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *ESWC (1)*, pages 154–169, 2011.

[30] O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL Queries over the Web of Linked Data. In *ISWC*, pages 293–309. 2009.

[31] O. Hartig and A. Langegger. A Database Perspective on Consuming Linked Data on the Web. *Datenbank-Spektrum*, 10(2):57–66, 2010.

[32] M. F. Husain, J. P. McGlothlin, M. M. Masud, L. R. Khan, and B. M. Thuraisingham. Heuristics-based query processing for large RDF graphs using Cloud Computing. *IEEE Trans. Knowl. Data Eng.*, 23(9):1312–1327, 2011.

[33] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, I. Miliaraki, M. Magiridou, and A. Papadakis-Pesaresi. Atlas: Storing, updating and querying RDF(S) data on top of DHTs. *J. Web Sem.*, 8(4):271–277, 2010.

[34] G. Ladwig and T. Tran. Linked Data Query Processing Strategies. In *ISWC*, pages 453–469, 2010.

[35] G. Ladwig and T. Tran. SIHJoin: Querying remote and local linked data. In *ESWC (1)*, pages 139–153, 2011.

[36] A. Langegger, W. Wöß, and M. Blöchl. A semantic web middleware for virtual data integration on the web. In *ESWC*, pages 493–507, 2008.

[37] J. J. Levandoski and M. F. Mokbel. RDF data-centric storage. In *ICWS*, pages 911–918, 2009.

[38] A. Matono, S. M. Pahlevi, and I. Kojima. RDFCube: A P2P-based three-dimensional index for structural joins on distributed triple stores. In *DBISP2P*, pages 323–330, 2006.

[39] N. Nakashole, M. Theobald, and G. Weikum. Scalable knowledge harvesting with high precision and high recall. In *WSDM*, pages 227–236, 2011.

[40] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984.

[41] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, pages 1137–1148, 2011.

[42] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[43] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, pages 627–640, 2009.

[44] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.

[45] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[46] T. W. Page, Jr. and G. J. Popek. Distributed management in local area networks. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '85, pages 135–142, New York, NY, USA, 1985. ACM.

[47] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, pages 383–392, 2004.

[48] M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, and G. Lausen. RDFPath: Path query processing on large RDF graphs with mapreduce. In *1st Workshop on High-Performance Computing for the Semantic Web (HPCSW 2011)*, 2011.

[49] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: scaling online social networks. *SIGCOMM Comput. Commun. Rev.*, 40:375–386, Aug. 2010.

[50] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez. Scaling Online Social Networks without Pains. In *Proceedings of NetDB 2009*, Oct. 2009.

[51] B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *ESWC*, pages 524–538, 2008.

[52] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.

[53] P. Ravindra, H. Kim, and K. Anyanwu. An intermediate algebra for optimizing RDF graph pattern matching on MapReduce. In *ESWC (2)*, pages 46–61, 2011.

[54] S. Sakr and G. Al-Naymat. Relational processing of RDF queries: a survey. *SIGMOD Record*, 38(4):23–28, 2009.

[55] S. Sarawagi. Information extraction. *Found. Trends databases*, 1:261–377, March 2008.

[56] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to pig latin. In *3rd International Workshop on Semantic Web Information Management (SWIM 2011)*, 2011.

[57] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[58] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *ISWC*, pages 601–616, 2011.

[59] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[60] F. M. Suchanek, M. Sozio, and G. Weikum. SOFIE: a self-organizing framework for information extraction. In *WWW*, pages 631–640, 2009.

[61] T. Teorey, L. Sam, N. Tom, and H. Jagadish. *Database Modelling and Design*. Morgan Kaufmann Publishers, third edition edition, 1999.

[62] B.-L. Tim, H. James, and L. Ora.

[63] G. Tsatsanifos, D. Sacharidis, and T. K. Sellis. On enhancing scalability for distributed RDF/S stores. In *EDBT*, pages 141–152, 2011.

[64] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11:37–57, 1985.

[65] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for Semantic Web data management. *PVLDB*, 1(1):1008–1019, 2008.

[66] K. Wilkinson. Jena property table implementations. Technical Report HPL-2006-140, Hewlett-Packard Labs, 2006.

[67] E. Wong and R. H. Katz. Distributing a database for parallelism. In *SIGMOD*, pages 23–29, 1983.

A Appendix

A.1 Experimental queries

Each dataset used for the experiments was tested against 20 queries, the first ten used to train the query load aware partitioner.

A.1.1 Sp2bench

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX sp: <http://localhost/vocabulary/bench/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dct: <http://purl.org/dc/terms/>
SELECT ?inproc ?author ?booktitle ?title ?proc ?ee ?page ?url ?yr ?
    abstract WHERE {
     ?inproc rdf:type sp:Inproceedings .
     ?inproc dc:creator ?author .
     ?inproc sp:booktitle ?booktitle .
     ?inproc dc:title ?title .
     ?inproc dct:partOf ?proc .
     ?inproc rdfs:seeAlso ?ee .
     ?inproc srwc:pages ?page .
     ?inproc foaf:homepage ?url .
     ?inproc dct:issued ?yr
     ?inproc sp:abstract ?abstract .
}
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX sp: <http://localhost/vocabulary/bench/>
PREFIX dct: <http://purl.org/dc/terms/>
SELECT ?yr WHERE {
  ?journal rdf:type sp:Journal .
  ?journal dc:title "Journal 1 (1940)"^^<http://www.w3.org/2001/
      XMLSchema#string> .
  ?journal dct:issued ?yr
}
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sp: <http://localhost/vocabulary/bench/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?name1 ?name2 WHERE {
  ?article1 rdf:type sp:Article .
  ?article2 rdf:type sp:Article .
  ?article1 dc:creator ?author1 .
  ?author1 foaf:name ?name1 .
  ?article2 dc:creator ?author2 .
  ?author2 foaf:name ?name2 .
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal
}

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sp: <http://localhost/vocabulary/bench/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?person ?name WHERE {
  ?article rdf:type sp:Article .
  ?article dc:creator ?person .
  ?inproc rdf:type sp:Inproceedings .
  ?inproc dc:creator ?person .
  ?person foaf:name ?name .
}

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sp: <http://localhost/vocabulary/bench/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?article ?pages WHERE {
  ?article rdf:type sp:Www .
  ?article swrc:pages ?pages .
}

PREFIX sp: <http://localhost/vocabulary/bench/>
SELECT ?abstract WHERE {
    <http://localhost/publications/articles/Journal1/1940/Article4>
        sp:abstract ?abstract .
}

PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?number1 ?editor WHERE{
    ?s swrc:number  ?number1.
    ?s swrc:volume "2"^^<http://www.w3.org/2001/XMLSchema#integer>
        .
    ?s swrc:editor ?editor .
}
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sp: <http://localhost/persons/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?isbn WHERE {
  sp:Paul_Erdoes swrc:editor ?article .
  ?article dc:references ?article2 .
  ?article2 swrc:isbn ?isbn .
}

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sp: <http://localhost/vocabulary/bench/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?name ?title ?journal WHERE {
  ?article rdf:type sp:Article .
  ?article dc:creator ?person .
  ?article dc:title ?title .
  ?article swrc:journal ?journal .
  ?person foaf:name ?name .
}

PREFIX sp: <http://localhost/vocabulary/bench/>
SELECT ?s WHERE {
    ?s sp:cdrom "http://www.oinked.tld/maxillary/dumpers.html"^^<
       http://www.w3.org/2001/XMLSchema#string> .
}

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sp: <http://localhost/vocabulary/bench/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?article WHERE {
    ?article rdf:type sp:Article .
    ?article swrc:month ?value .
}

SELECT ?ee WHERE {
  ?publication <http://www.w3.org/2000/01/rdf-schema#seeAlso> ?ee
}

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?predicate WHERE {
  ?person rdf:type foaf:Person .
  ?subject ?predicate ?person .
}
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
SELECT ?yr ?name ?document WHERE {
    ?class rdfs:subClassOf foaf:Document .
    ?document rdf:type ?class .
    ?document dct:issued ?yr .
    ?document dc:creator ?author .
    ?author foaf:name ?name .
}

SELECT ?subject ?predicate WHERE { ?subject ?predicate <http://
   localhost/persons/Paul_Erdoes> . }

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?name WHERE {
  ?erdoes rdf:type foaf:Person .
  ?erdoes foaf:name "Paul Erdoes"^^<http://www.w3.org/2001/
     XMLSchema#string> .
  ?document dc:creator ?erdoes.
  ?document dc:creator ?author.
  ?author foaf:name ?name.
} LIMIT 352

PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?number1 ?editor WHERE{
  ?s swrc:number ?number1.
  ?s swrc:volume ?number .
  ?s swrc:editor ?editor .
}

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX sp: <http://localhost/vocabulary/bench/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
SELECT ?title WHERE {
  ?article rdf:type sp:Article .
  ?article dct:references ?article2 .
  ?article2 dc:title ?title .
}

PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
PREFIX dct: <http://purl.org/dc/terms/>
SELECT ?title WHERE {
  ?article dc:title ?title .
  ?article dc:references ?article2 .
  ?article2 dc:title "cheaters kerneling napoleons"^^<http://www.w3
     .org/2001/XMLSchema#string> .
}

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?document WHERE {
  ?class rdfs:subClassOf foaf:Document .
  ?document rdf:type ?class .
  ?document dct:issued ?yr .
  ?document dct:partOf <http://localhost/publications/procs/
     Proceeding1/1954> .
}
```

A.1.2 Berlin20K

```
SELECT ?subject ?property WHERE {
  ?subject ?property <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
     v01/instances/dataFromVendor100/Vendor100>
}
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
SELECT ?product ?label WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type bsbm:Product
}
```

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?feature WHERE{
  ?product a ?ProductType ; bsbm:productFeature ?feature .
  ?offer bsbm:product ?product ; bsbm:price ?price .
}
```

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?reviewer ?country WHERE {
  ?review bsbm:reviewFor ?product ; rev:reviewer ?reviewer .
  ?reviewer bsbm:country ?country .
}
```

```
PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/dataFromProducer3/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?label ?propertyTextual2 ?propertyTextual1 WHERE {
  ?prod rdfs:label ?label .
  ?prod rdfs:comment ?comment .
  ?prod bsbm:producer ?p .
  ?p rdfs:label ?producer .
  ?prod bsbm:productPropertyTextual1 ?propertyTextual1 .
  ?prod bsbm:productPropertyTextual2 ?propertyTextual2 .
}
```

```
PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?product ?label ?propertyTextual WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type bsbminst:ProductType1 .
  ?product bsbm:productFeature bsbminst:ProductFeature1 .
  ?product bsbm:productFeature bsbminst:ProductFeature2 .
  ?product bsbm:productPropertyTextual1 ?propertyTextual .
  ?product bsbm:productPropertyNumeric1 ?p1 .
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?vendor WHERE {
  ?product a ?ProductType .
  ?offer bsbm:product ?product .
  ?offer bsbm:vendor ?vendor .
  ?offer bsbm:price ?price .
}

PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?product ?label WHERE {
  ?product rdfs:label ?label . ?product a bsbminst:ProductType1 .
  ?product bsbm:productFeature bsbminst:ProductFeature1 .
  ?product bsbm:productFeature bsbminst:ProductFeature2 .
  ?product bsbm:productPropertyNumeric1 ?value1 .
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?offer ?price WHERE {
  ?offer bsbm:product <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
     v01/instances/dataFromProducer1/Product1> .
  ?offer bsbm:vendor ?vendor . ?offer dc:publisher ?vendor .
```

```
  ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#
     US> .
  ?offer bsbm:deliveryDays ?deliveryDays .
  ?offer bsbm:price ?price .
  ?offer bsbm:validTo ?date
}

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
SELECT ?product ?label WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type bsbm:Product .
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?offer ?price WHERE {
  ?offer bsbm:product <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
     v01/instances/dataFromProducer1/Product1> .
  ?offer bsbm:vendor ?vendor .
  ?offer dc:publisher ?vendor .
  ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#
     US> .
  ?offer bsbm:deliveryDays ?deliveryDays .
  ?offer bsbm:price ?price .
  ?offer bsbm:validTo ?date .
}

SELECT ?property ?hasValue WHERE {
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
     dataFromVendor100/Vendor100> ?property ?hasValue .
}

PREFIX rev: <http://purl.org/stuff/rev#>
SELECT ?x WHERE {
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
     dataFromRatingSite1/Review1> rev:reviewer ?x
}

SELECT * WHERE {
  ?s <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
     vendor> ?o .
}
```

```
PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?product ?label ?page WHERE{
  ?product a bsbminst:ProductType84 .
  ?product bsbm:producer ?producer .
  ?producer foaf:homepage ?page .
  ?producer bsbm:country ?country1 .
  ?producer rdfs:label ?label .
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?vendor WHERE {
  ?product a ?ProductType .
  ?offer bsbm:product ?product .
  ?offer bsbm:vendor ?vendor .
  ?offer bsbm:price ?price .
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
SELECT ?otherProduct WHERE {
  ?otherProduct a bsbm:Product .
  ?otherProduct bsbm:productFeature ?otherFeature .
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT DISTINCT ?offer ?price WHERE {
  ?offer bsbm:product <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
     v01/instances/dataFromProducer1/Product1> .
  ?offer bsbm:vendor ?vendor .
  ?offer dc:publisher ?vendor .
  ?vendor bsbm:country ?country .
  ?offer bsbm:deliveryDays ?deliveryDays .
  ?offer bsbm:price ?price .
  ?offer bsbm:validTo ?date .
}
```

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?reviewer ?country WHERE {
  ?review bsbm:reviewFor ?product ; rev:reviewer ?reviewer .
  ?reviewer bsbm:country <http://downlode.org/rdf/iso-3166/
     countries#US>.
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?reviewer WHERE {
  ?product bsbm:producer ?Producer .
  ?review bsbm:reviewFor ?product .
  ?review rev:reviewer ?reviewer .
  ?review bsbm:rating1 ?score .
}
```

A.1.3 Berlin500K

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
SELECT ?productType WHERE{
  ?productType a bsbm:ProductType .
  ?product a ?productType .
  ?product bsbm:producer ?producer .
  ?producer bsbm:country ?country1 .
  ?review bsbm:reviewFor ?product .
  ?review rev:reviewer ?reviewer .
  ?reviewer bsbm:country ?country2 .
}
```

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
SELECT ?otherProduct WHERE {
  ?otherProduct a bsbm:Product .
  ?otherProduct bsbm:productFeature ?otherFeature .
}
```

```
PREFIX rev: <http://purl.org/stuff/rev#>
SELECT ?x WHERE {
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
     dataFromRatingSite1/Review6> rev:reviewer ?x .
}
```

```
SELECT * WHERE {
  ?s <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
     vendor> ?o .
}
```

```
PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?product ?label ?propertyTextual WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type bsbminst:ProductType1 .
  ?product bsbm:productFeature bsbminst:ProductFeature1 .
  ?product bsbm:productFeature bsbminst:ProductFeature2 .
  ?product bsbm:productPropertyTextual1 ?propertyTextual .
  ?product bsbm:productPropertyNumeric1 ?p1 .
}
```

```
PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
SELECT ?productType WHERE{
  ?productType a bsbminst:ProductType1 .
  ?product a ?productType .
  ?product bsbm:producer ?producer .
  ?producer bsbm:country ?country1 .
  ?review bsbm:reviewFor ?product .
  ?review rev:reviewer ?reviewer .
  ?reviewer bsbm:country ?country2 .
}

PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
SELECT * WHERE {
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
     dataFromProducer2/Product69> bsbm:productFeature ?otherFeature
      .
  ?otherFeature <http://purl.org/dc/elements/1.1/publisher> ?o .
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
SELECT * WHERE {
  ?s <http://xmlns.com/foaf/0.1/homepage> ?o .
}

PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?product ?label ?propertyTextual WHERE {
  ?product rdfs:comment ?label .
  ?product rdf:type bsbminst:ProductType12 .
  ?product bsbm:productFeature ?feature .
  ?product bsbm:productPropertyTextual2 ?propertyTextual .
  ?product bsbm:productPropertyNumeric1 ?p1 .
}
```

```
PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
SELECT ?product ?label ?page WHERE{
  ?product a bsbminst:ProductType84 .
  ?product bsbm:producer ?producer .
  ?producer <http://xmlns.com/foaf/0.1/homepage> ?page .
  ?producer bsbm:country ?country1 .
  ?producer <http://www.w3.org/2000/01/rdf-schema#label> ?label .
}

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
SELECT ?product ?label WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type bsbm:Product
} LIMIT 10000

PREFIX bsbminst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?sum WHERE {
  ?reviewer foaf:mbox_sha1sum ?sum .
  ?review bsbm:reviewFor ?product .
  ?review rev:reviewer ?reviewer .
  ?reviewer bsbm:country <http://downlode.org/rdf/iso-3166/
     countries#DE> .
}

PREFIX rev: <http://purl.org/stuff/rev#>
SELECT ?x WHERE {
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
     dataFromRatingSite1/Review3> rev:reviewer ?x
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
SELECT DISTINCT ?offer ?price WHERE {
  ?offer bsbm:product <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
     v01/instances/dataFromProducer1/Product1> .
  ?offer bsbm:vendor ?vendor .
  ?offer dc:publisher ?vendor .
  ?vendor bsbm:country ?country .
  ?offer bsbm:deliveryDays ?deliveryDays .
  ?offer bsbm:price ?price .
  ?offer bsbm:validTo ?date .
}

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?reviewer WHERE {
  ?review bsbm:reviewFor ?product ; rev:reviewer ?reviewer .
  ?reviewer bsbm:country <http://downlode.org/rdf/iso-3166/
     countries#US>.
} LIMIT 1000

PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?reviewer WHERE {
  ?product bsbm:producer ?Producer .
  ?review bsbm:reviewFor ?product .
  ?review rev:reviewer ?reviewer .
  ?review bsbm:rating1 ?score .
} LIMIT 10000

SELECT ?subject ?property WHERE {
  ?subject ?property <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
     v01/instances/dataFromVendor100/Vendor100>
}

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
SELECT ?product ?label WHERE {
  ?product rdfs:label ?label .
  ?product rdf:type bsbm:Product
}
```

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
   vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?vendor WHERE {
  ?product a ?ProductType .
  ?offer bsbm:product ?product .
  ?offer bsbm:vendor ?vendor .
  ?offer bsbm:price ?price .
}

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rev: <http://purl.org/stuff/rev#>
SELECT ?text WHERE {
  ?person foaf:name "Ruggiero-Delane" .
  ?person rev:reviewer ?review .
  ?review rev:text ?text .
}
```

A.1.4 Billiontriplechallenge

```
SELECT ?lat ?long WHERE {
  ?a [] "Eiffel Tower".
  ?a <http://www.geonames.org/ontology#inCountry> "http://www.
     geonames.org/countries/#FR".
  ?a <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat.
  ?a <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long.
}

SELECT ?b ?p ?bn WHERE {
  ?a [] "Tim Berners-Lee".
  ?a <http://dbpedia.org/property/dateOfBirth> ?b.
  ?a <http://dbpedia.org/property/placeOfBirth> ?p.
  ?a <http://dbpedia.org/property/name> ?bn.
}

SELECT ?t ?lat ?long WHERE {
  ?a <http://dbpedia.org/property/wikilink> <http://dbpedia.org/
     resource/List_of_World_Heritage_Sites_in_Europe> .
  ?a <http://dbpedia.org/property/title> ?t .
  ?a <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat .
  ?a <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long .
  ?a <http://dbpedia.org/property/wikilink> <http://dbpedia.org/
     resource/Middle_Ages> .
}

SELECT * WHERE {
  ?p <http://dbpedia.org/property/name> "Krebs, Emil".
  ?p <http://dbpedia.org/property/deathPlace> ?l.
  ?c <http://www.geonames.org/ontology#name> ?l.
  ?c [] ?l.
  ?c <http://www.geonames.org/ontology#featureClass> <http://www.
     geonames.org/ontology%23P>.
  ?c <http://www.geonames.org/ontology#inCountry> "http://www.
     geonames.org/countries/#DE".
  ?c <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long.
  ?c <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat.
}

SELECT DISTINCT ?l ?long ?lat WHERE {
  ?a [] "Barack Obama".
  ?a <http://dbpedia.org/property/placeOfBirth> ?l.
  ?l <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat.
  ?l <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long.
}
```

```
SELECT DISTINCT ?d WHERE {
  ?a <http://dbpedia.org/property/senators> ?c.
  ?a <http://dbpedia.org/property/name> ?d.
  ?c <http://dbpedia.org/property/profession> <http://dbpedia.org/
     resource/Veterinarian>.
  ?a <http://www.w3.org/2002/07/owl#sameAs> ?b.
  ?b <http://www.geonames.org/ontology#inCountry> "http://www.
     geonames.org/countries/#US".
}

SELECT DISTINCT ?a ?b ?lat ?long WHERE {
  ?a <http://dbpedia.org/property/spouse> ?b.
  ?a <http://dbpedia.org/property/wikilink> ?x.
  ?a <http://dbpedia.org/property/placeOfBirth> ?c.
  ?b <http://dbpedia.org/property/placeOfBirth> ?c.
  ?c <http://www.w3.org/2002/07/owl#sameAs> ?c2.
  ?c2 <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat.
  ?c2 <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long.
}

SELECT ?a ?y WHERE {
  ?a <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
     dbpedia.org/class/yago/Politician110451263>.
  ?a <http://dbpedia.org/property/years> ?y.
  ?a <http://xmlns.com/foaf/0.1/name> ?n.
  ?b <http://purl.org/dc/elements/1.1/subject> ?n.
  ?b [] ?n.
  ?b <http://purl.org/dc/elements/1.1/subject> "Blackwater".
}

SELECT ?desc ?link ?agent ?descag WHERE{
  ?s a <http://xmlns.com/foaf/0.1/Document> .
  ?s a <http://purl.org/rss/1.0/item> .
  ?s  <http://purl.org/rss/1.0/description> ?desc .
  ?s <http://purl.org/rss/1.0/link> ?link .
  ?link <http://webns.net/mvcb/generatorAgent> ?agent .
  ?agent  <http://purl.org/dc/elements/1.1/description> ?descag
} LIMIT 10

SELECT ?prop ?prop2 WHERE {
  ?prop <http://www.w3.org/2000/01/rdf-schema#domain> <http://xmlns
     .com/foaf/0.1/Person> .
  ?prop <http://www.daml.org/2001/03/daml+oil#inverseOf> ?prop2 .
}

SELECT ?label ?label1 ?creator WHERE {
```

```
  ?s <http://www.w3.org/2000/01/rdf-schema#label> ?label .
  ?obj a <http://xmlns.com/foaf/0.1/Document> .
  ?s <http://www.w3.org/2000/01/rdf-schema#seeAlso> ?obj .
  ?obj <http://www.w3.org/2000/01/rdf-schema#label> ?label1 .
  ?obj <http://purl.org/dc/elements/1.1/creator> ?creator .
}

SELECT ?s WHERE {
  ?s <http://purl.org/dc/elements/1.1/article> ?prop  .
}

SELECT * WHERE{
  ?s <http://purl.org/dc/elements/1.1/contributor> "testing" .
  ?s <http://purl.org/dc/elements/1.1/format> ?format .
  ?s <http://purl.org/dc/elements/1.1/date> ?date .
  ?s <http://purl.org/dc/elements/1.1/coverage> ?cov .
  ?s <http://purl.org/dc/elements/1.1/language> ?lang .
  ?s <http://purl.org/dc/elements/1.1/creator> ?creator .
  ?s <http://purl.org/dc/elements/1.1/publisher> ?pub .
  ?s <http://purl.org/dc/elements/1.1/relation> ?rel .
  ?s <http://purl.org/dc/elements/1.1/subject> ?sub .
}

SELECT * WHERE{
  ?s <http://purl.org/dc/elements/1.1/contributor> ?contr .
  ?s <http://purl.org/dc/elements/1.1/format> ?format .
  ?s <http://purl.org/dc/elements/1.1/date> ?date .
  ?s <http://purl.org/dc/elements/1.1/coverage> ?cov .
  ?s <http://purl.org/dc/elements/1.1/language> ?lang .
  ?s <http://purl.org/dc/elements/1.1/creator> ?creator .
  ?s <http://purl.org/dc/elements/1.1/publisher> ?pub .
  ?s <http://purl.org/dc/elements/1.1/relation> ?rel .
  ?s <http://purl.org/dc/elements/1.1/subject> ?sub .
}

SELECT * WHERE {
  ?p <http://dbpedia.org/property/name> ?name.
  ?p <http://dbpedia.org/property/deathPlace> ?l.
  ?c <http://www.geonames.org/ontology#name> ?l.
  ?c [] ?l.
  ?c <http://www.geonames.org/ontology#featureClass> <http://www.
     geonames.org/ontology%23P>.
  ?c <http://www.geonames.org/ontology#inCountry> ?country .
  ?c <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?long.
  ?c <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?lat.
} LIMIT 1000
```

```
SELECT ?desc ?link ?agent ?descag WHERE{
  ?s a <http://xmlns.com/foaf/0.1/Document> .
  ?s a <http://purl.org/rss/1.0/image> .
  ?s  <http://purl.org/rss/1.0/description> ?desc .
  ?s <http://purl.org/rss/1.0/link> ?link .
  ?link <http://webns.net/mvcb/generatorAgent> ?agent .
  ?agent  <http://purl.org/dc/elements/1.1/description> ?descag
}

SELECT DISTINCT ?name ?name2 WHERE {
  ?person <http://dbpedia.org/property/religion> <http://dbpedia.
     org/resource/Roman_Catholic_Church> .
  ?person <http://xmlns.com/foaf/0.1/page> ?page .
  ?person <http://dbpedia.org/property/placeOfDeath> ?pdeath .
  ?person <http://dbpedia.org/property/predecessor> ?prede .
  ?person <http://xmlns.com/foaf/0.1/img> ?img .
  ?person <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
  ?person <http://dbpedia.org/property/ordination> ?ordination .
  ?person <http://dbpedia.org/property/title> ?title .
  ?person <http://www.w3.org/2000/01/rdf-schema#label> ?name .
  ?person <http://dbpedia.org/property/name> ?name2 .
}

SELECT ?name WHERE {
  ?s <http://dbpedia.org/property/team> "retired" .
  ?s <http://dbpedia.org/property/name> ?name .
}

SELECT * WHERE {
  ?person a <http://xmlns.com/foaf/0.1/Person> .
  ?person <http://dbpedia.org/property/spouse> ?spouse .
  ?person <http://dbpedia.org/property/placeOfDeath> ?city .
  ?spouse <http://dbpedia.org/property/placeOfDeath> ?city .
  ?country <http://dbpedia.org/property/capital> ?city
} LIMIT 10

SELECT * WHERE {
  ?x <http://dbpedia.org/property/wikilink> ?link .
  ?x <http://dbpedia.org/property/disambiguates> ?y .
  ?y a ?sth }
```

A.2 Optimal Allocation

The problem of optimal allocation of fragments in PARTOUT can be formulated as an integer linear program. Let QL be a representative query load for some data set T, M a non-redundant and complete fragmentation of T output by our query load aware partitioner, and $G(QL, M)$ the global fragment graph. Define a set of $|M|^2$ constants ω_{ij}, which denote the weight of the edge between fragments m_i and m_j in $G(QL, M)$. This value represents the number of times fragments m_i and m_j join in the query load QL. If there is no edge between m_i and m_j, then $\omega_{ij} = 0$. Define $E = \sum_{i=1}^{|M|} \sum_{j=1}^{|M|} \omega_{ij}$. We want to assign the fragments in M to a set of hosts H with $|H| = n$ such that joining fragments are assigned to the same host but the total load in the system is fairly distributed. The load $L(m_i)$ induced by a fragment m_i, is $L(m_i) = s(m_i)f(m_i)$ where $s(m_i)$ is its size and $f(m_i)$ denotes its frequency of access. The total load L in the system is:

$$L = \sum_{i=0}^{|M|} L(m_i)$$

Define the set of $|M|n$ boolean variables x_{ij} such that:

$$x_{ih} = \begin{cases} 1; \text{ Fragment } m_i \text{ is assigned to host } h \\ 0; \text{ otherwise} \end{cases}$$

We can formulate the optimal allocation problem as an integer linear program:

$$\min \sum_{i=1}^{|M|} \sum_{j=1; j \neq i}^{|M|} (E - \omega_{ij}) \sum_{h=1}^{n} (x_{ih} + x_{jh})$$

under the following constrains:

(a) A fragment must be assigned to only host

$$\forall\ i \in \{1, 2, \ldots |M|\}\ \sum_{h=1}^{n} x_{ih} = 1$$

(b) The load among the hosts must be balanced

$$\forall\ h \in \{1, 2, \ldots n\}\ \sum_{i=1}^{|M|} s(m_i) f(m_i) x_{ih} \leq \frac{L}{|H|}$$

(c) Integrality constraints

$$\forall\ i, h\ x_{ih} \in \{0, 1\}$$