Query Processing for RDF Databases

Zoi Kaoudi[1] and Anastasios Kementsietsidis[2]

[1] IMIS, Athena Research Center, Athens, Greece
zoi@imis.athena-innovation.gr
[2] Google Research, Mountain View, USA
akement@google.com

Abstract. RDF has become recently a very popular data model used in a variety of applications and use cases in both academia and industry. Query processing and evaluation is a central component in data management in general and is, thus, unsurprisingly one of the most active areas of research in the field of RDF data management. In this chapter we provide an overview of query processing techniques for the RDF data model using different system architectures. We survey techniques for both centralized and distributed RDF stores, including peer-to-peer, federated and cloud-based systems.

1 Introduction

Query processing and evaluation is such a central component of data management that unsurprisingly is one of the most active areas of research in this field. As databases evolve and new architectures or models emerge, efficient query processing becomes probably the most pressing problem to address in these new environments. One has to just consider the move from centralized to distributed [59] or streaming [24] sources, or the emergence of new models like XML [60] or RDF [62] and notice the corresponding jump in research works in conferences and workshops on this topic. In spite of this large body of past works, there is always room for improvement or for the generation of new techniques. Each new environment (be it an architecture or a model) carries its own set of assumptions and requirements that necessitate a revision of past query processing techniques, or in the worst case lead to the development of new ones.

In what follows, we provide an overview of query processing techniques in the context of the RDF data model and the SPARQL query language [31]. To put things in perspective and present techniques and architectures in an organized fashion, we are going to use the virtual system overview of Figure 1 as a guide to our presentation. Our system consists of two layers, a *decentralized* layer and a *individual source* layer. There are many architectures that one can use to build a decentralized layer for RDF sources, and in the figure we list the three we are covering in this chapter, namely, *federated*, *peer-to-peer*, and *cloud* systems. Any of these system architectures assumes that it is built on top of a set of RDF sources where each source in the set provides, as a minimum requirement, a data access entry, e.g., a SPARQL endpoint service. Now, internally each individual

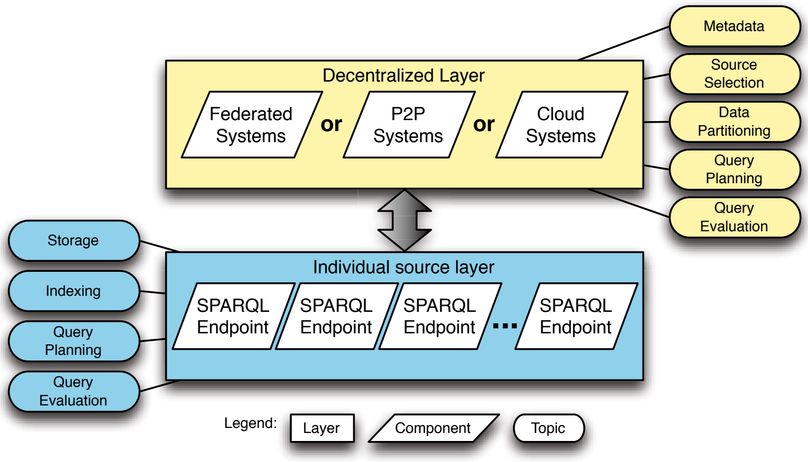

Fig. 1. Virtual RDF system overview

RDF source can have its own architecture which can range from a native RDF store, to a relational store appropriately configured to store RDF data, or even to a relational store with relational data served as RDF through a front-end. Irrespectively of the particular architecture used in either layer, there are a number of research topics that are common across the architectures within each layer, and we list those adjacent to each layer in the figure. Though this is clearly not an exhaustive list of topics, they constitute the minimal topics a system should address to have a workable solution in this space. We briefly review the relevance and importance of each topic in the following paragraphs.

1.1 Decentralized Layer Topics

– Metadata: In decentralized systems, there is a well-known trade-off between query performance and source autonomy. At one end of the spectrum, one can consider a system where the sources only provide a SPARQL endpoint service and no other metadata (e.g., statistics, indexes) is known about them [75]. Since query processing relies heavily on knowledge about such metadata, when no such information is available to a decentralized system, the system must try to dynamically compute the necessary metadata. This process is bound to have an impact on end-to-end query performance since additional work must be performed before the processing of the actual query is initiated. At the other end of the spectrum, one can consider a system with tighter control of the sources where each source is required to generate appropriate metadata and provide them to the decentralized system [4, 28]. This clearly facilitates query processing but requires that the sources themselves are willing to do work for the decentralized system (in addition to query answering) by regularly updating their metadata to reflect the changes in their data.

– Source selection: Not all sources in a decentralized system contain data that are relevant to the query at hand. Figuring out which sources are relevant is another factor affecting query performance since incorrectly identified sources might incur additional costs (both in terms of processing time and communication overhead) without contributing to the end result. There is a close relationship between the topic of source selection and that of metadata, since often the latter is used to provide an answer for the former.
– Data partitioning: Another area where source autonomy is a key factor is in the distribution of data across the decentralized system. At one extreme one can envision a federated system in which data partitioning comes naturally as sources join the federation with their own data. At the other extreme, one can envision a P2P-style system in which sources join the system and it is the decentralized layer that is responsible for assigning which source is going to store which data. While there is not much to be said about partitioning in the former scenario, in the latter scenario partitioning requires an algorithm to partition and distribute the data evenly across the sources. Ideally, other than data load, the algorithm should aim to minimize communication costs during query evaluation.
– Query planning: Once the sources relevant for a query have been identified, and relevant metadata have been computed or retrieved, query planning is a common next step. The type of planning available to a decentralized system also depends on the level of autonomy of the underlying sources. In one extreme case, one can think of a system with fairly uncooperative sources where the sources only provide a SPARQL endpoint service. In such a setting, the results from individual sources must be sent to a central location where the actual query evaluation takes place. It is not hard to see that in such a setting the communication cost may far outweigh actual query evaluation costs. In a more collaborative setting, the sources of the decentralized system might be willing to cooperate in performing distributed query evaluation. Query planning becomes more central at this setting since it specifies the order (as in the centralized case) with which results would be computed and propagated.
– Query evaluation: More often than not, there are multiple ways to combine results from different sources to compute an answer to an input query. Each of these ways generates its own plan and results in a query. All these queries must be evaluated to guarantee the *completeness* of query answers (i.e., that all the results of the query are retrieved and none is *missed*). Executing all these queries (and in parallel) poses its own challenges. Clearly, one can evaluate each of these queries independently, but when there is sharing of sub-queries across the different queries, there is the potential for optimization [50] (e.g., by executing these common sub-queries only once and sharing their results across the different plans).

1.2 Individual Source Layer Topics

- Storage: The storage strategy, or how an individual RDF source decides to internally represent RDF data, is a central topic which influences every aspect of the source, from indexing, to planning and evaluation. There are multiple alternatives here with some sources using novel *native* representations for RDF data (e.g., Jena TDB [39], RDF-3X [58]), while others using relational databases as the back-end (e.g., Jena SDB [39], DB2RDF [17], C-Store [1]) and designing appropriate schemas to store the RDF data.
- Indexing: Indexing plays a key role in query evaluation in general but in the context of RDF this role becomes even more central since indexes sometimes morph into an actual storage strategy (e.g., Jena TDB uses a custom implementation of B+-trees as central component of its storage strategy). This interplay between storage and indexes results in some novel strategies in terms of query evaluation (e.g., [58]).
- Query planning: Unlike query planning in the decentralized setting which focuses on reducing communication costs (mostly through join ordering), planning at the individual source level is typically more complex and has a larger search space of alternatives to consider. One reason for this added complexity comes from the multitude of access methods that are usually available to the planner, which opens up the way for accessing source data and combining intermediate query results using a host of techniques.
- Query evaluation: In traditional DBMSs, there is a tight coupling between the planner and the query evaluation module in the sense that they are both part of the same system (and are often developed by the same group of people). However, this is not always the case with RDF sources, and especially with those that use relational databases as the back-end. In such sources, the RDF query planner might be built as a separate component on top of any existing relational back-end (e.g., see Jena SDB [39] or DB2RDF [17]). Since planning happens outside the relational engine, the planner can dictate the order with which different sub-queries are executed, but it has little control over the actual evaluation by the back-end. Therefore, in such settings, the efficiency of query evaluation is largely determined by how well the planner can predict the efficiency of queries before these are evaluated by the underlying relational store.

Using Figure 1 as a guide for our presentation, we organize the remainder of this chapter in the following manner. For completeness, the next section presents some necessary preliminaries in terms of the RDF model and the SPARQL query language. Then in Section 3, we provide an overview of existing single-source systems that constitute our building blocks in Figure 1 towards constructing any of the decentralized systems shown there. Sections 4, 5 and 6 present respectively existing P2P, federated and cloud-based systems. Not all the topics we covered briefly in the introduction are of equal importance for each presented system and not all systems provide novel solutions across all the topics. Therefore throughout the sections as we describe a (single-source or decentralized) system we make certain that we highlight the topics that are of most notable and novel about the system. The paper concludes with a summary of the work and a short discussion regarding open research directions in the field of RDF query processing.

2 Preliminaries

In this section we introduce the Resource Description Framework (RDF) together with its accompanying schema language RDFS. In addition, we present SPARQL, the standardized query language for querying RDF and RDFS data.

2.1 RDF

The emergence of RDF originated from the Semantic Web vision, where anything in the Web can be interpreted by machines, converting the *Web of documents* to a *Web of data*. RDF is the main data model used to achieve this goal.

RDF offers the following basic constructs:

– *URIs*: Universal Resource Identifiers are used to represent *resources*, e.g., a Web page, a book, an author, a paper or a computer file. In addition, URIs are used to identify *properties*, which are the attributes of a resource or connect two resources. For example, `hasName` may be used for describing the name of an author or `isAuthorOf` may be used to connect a book resource with an author resource.
– *Literals*: Literals are constant values of any property. For example, `"John Doe"` may be the value of the property `hasName`.
– *Blank nodes*: Blank nodes are anonymous resources which are not expressed by a URI but in the form of `_:bnodeID`. The purpose of blank nodes is to encode n-ary relationships and/or to make statements about resources that might not have global URIs but can be described in terms of their relationship with other resources.

Facts in RDF are represented by sets of *triples*. Each triple consists of the resource the fact is about (subject), the property of the resource the statement refers to (predicate), and the value of that property (object). Another representation of RDF data is that of *labelled graphs* where the subject and object is depicted as a *node* and the property as a *directed labelled edge* from the subject node to the object node labelled by the predicate name. More formally:

Definition 1 (RDF triple). *Let U, L and B denote three pairwise disjoint sets of URIs, literals, and blank nodes, respectively. A* triple *is a tuple (s, p, o) from $(U \cup B) \times U \times (U \cup L \cup B)$, where s is the subject, p is the predicate (a.k.a. property) and o is the object of the triple.*

Definition 2 (RDF term and element). *An* RDF term *is any value from $(U \cup L \cup B)$ and an* RDF element *is any among the subject, predicate and object of an RDF triple.*

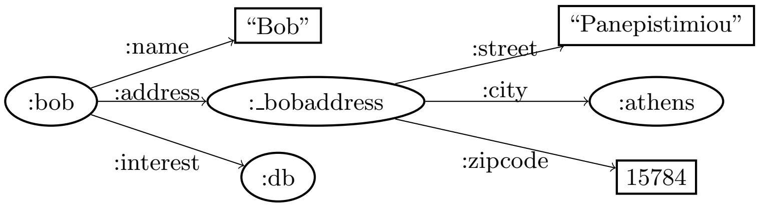

Fig. 2. RDF graph example

Definition 3 (RDF graph). *An RDF graph is a set of RDF triples.*

For example, the following set of triples state that the resource `http://tiny/bob` named `"Bob"` has as research interest databases and lives in a specified address in the city of Athens. The constant `:_bobaddress` denotes a blank node to describe the address of Bob.

```
http://tiny/bob http://tiny/name "Bob" .
http://tiny/bob http://tiny/interest http://tiny/db .
http://tiny/bob http://tiny/address :_bobaddress .
:_bobaddress http://tiny/street "Panepistimiou" .
:_bobaddress http://tiny/postalCode 15784 .
:_bobaddress http://tiny/city http://tiny/athens .
```

Figure 2 shows the same RDF information in an RDF graph. We depict a resource with an oval circle and a literal with a rectangle. The labeled arcs represent the properties of the RDF graph. For clarity reasons, we attach the empty prefix '`:`' for all URIs to distinguish them from the corresponding literals.

2.2 RDF Schema (RDFS)

The RDF data model offers a simple way for describing relationships among resources in terms of named properties and values, but does not provide mechanisms for declaring these properties, nor does it provide ways for defining the relationships between these properties and other resources. This is the role of RDF Schema (RDFS) [18]. RDFS is the vocabulary of RDF; it provides the means to a user to define terms that will be used in RDF statements and give specific meaning to them. RDFS defines not only the properties of a resource (e.g., title, author, subject etc.) but also the kinds of resources being described (people, paper, Web pages, books etc.). Resources having similar characteristics can be divided into groups called RDFS *classes*. We will refer to both RDF and RDFS information with the term RDF(S).

Figure 3 shows an RDF(S) graph which defines the relationships between classes and assigns the resource `http://tiny/bob` to a certain class. The property `rdfs:subClassOf` is a predefined property in RDFS and denotes a specialization relationship between two classes. For example, `http://tiny/graduateStudent` is a

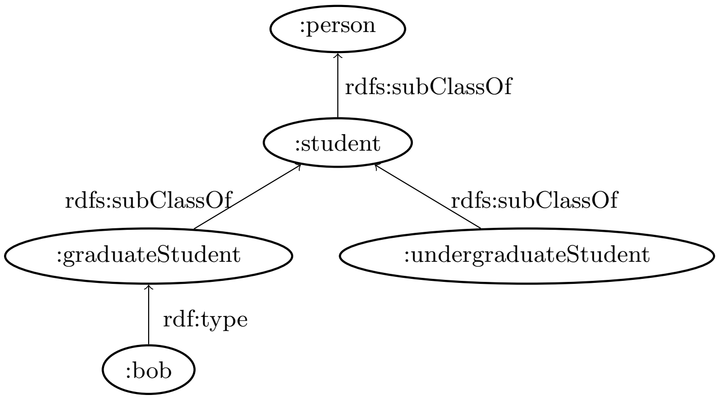

Fig. 3. An RDF(S) graph example

class which defines a specialized type of class `http://tiny/student`. Another predefined property commonly used in RDF is `rdf:type`. This property defines the members of a certain class. For example, resource `http://tiny/bob` is a member of class `http://tiny/graduateStudent`. The members of a class are also known as *instances* of the class. Namespaces `rdf` and `rdfs` are the namespaces of the core RDF and RDFS vocabulary defined by the URIs `http://www.w3.org/1999/02/22-rdf-syntax-ns#` and `http://www.w3.org/2000/01/rdf-schema#` respectively.

Another feature of RDFS is that it also provides vocabulary for describing how properties and classes are intended to be connected in an RDF(S) graph. The most important information of this kind is supplied by using the RDF Schema properties `rdfs:domain` and `rdfs:range`. The `rdfs:domain` property is used to indicate that a particular property applies to a specific class. The `rdfs:range` property is used to indicate that the values of a particular property are instances of a specific class or are types of specific literals. For example, in the RDF graph of Figure 2 we could add the following RDFS triples for properties `http://tiny/name` and `http://tiny/interest`:

```
http://tiny/name rdfs:domain http://tiny/person .
http://tiny/interest rdfs:domain http://tiny/student .
http://tiny/name rdfs:range string .
```

This means that the property `http://tiny/name` applies to instances of class `http://tiny/person`, while the property `http://tiny/interest` applies to instances of class `http://tiny/student`. In addition, the last triple states that property `http://tiny/name` takes values that are strings.

The most important functionality of RDFS is the ability to make inferences using the RDFS entailment rules [32]. This means that one can *infer new triples* from an RDF schema and the set of rules. Such triples are often called *implicit or inferred* triples. For example, the RDFS property `rdfs:subClassOf` is defined as a transitive property in the rule $rdfs11$ of the RDFS Semantics [32]:

$$\frac{\texttt{(a, rdfs:subClassOf, b)}, \texttt{(b, rdfs:subClassOf, c)}}{\texttt{(a, rdfs:subClassOf, c)}}$$

This rule says that if class `a` is a subclass of class `b` and class `b` is a subclass of class `c`, then we infer that class `a` is a subclass of class `c`. Thus, in the example of Figure 3, we can infer that class `:graduateStudent` is a subclass of class `:person`. Then, using this inferred triple and rule $rdfs9$ from [32]:

$$\frac{\texttt{(a, rdfs:subClassOf, b)}, \texttt{(r, rdf:type, a)}}{\texttt{(r, rdf:type, b)}}$$

we infer that `http://tiny/bob` in Figure 3 is also an instance of `http://tiny/person`. The complete set of the RDFS entailment rules can be found in [32].

2.3 SPARQL

SPARQL [31] is the standard query language for RDF recommended by W3C and has the ability to extract information about both the data and the schema. A core concept of SPARQL is that of a triple pattern, which is a triple with the possibility of a variable in any of the subject, predicate or object positions. A query that contains a conjunction of triple patterns is called *basic graph pattern* (BGP) query, and is the conjunctive fragment of SPARQL allowing to express the core select-project-join queries. More formally:

Definition 4 (Triple pattern). *Let U, L, B and V denote the pairwise disjoint sets of URIs, literals, blank nodes, and variables respectively. A* triple pattern *is a tuple* (s, p, o) *from* $(U \cup B \cup V) \times (U \cup V) \times (U \cup L \cup B \cup V)$.

Definition 5 (Basic Graph Pattern). *A* basic graph pattern *query is a conjunction of triple patterns.*

A basic graph pattern matches a subgraph of the RDF data when RDF terms from the subgraph can be substituted with the variables of the graph pattern.

The syntax of SPARQL follows an SQL-like `select-from-where` paradigm. The `select` clause specifies the variables that should appear in the query results. Each variable in SPARQL is prefixed with `?`. The `from` clause specifies the RDF(S) graph that should be used for answering the query. If not used, the query runs over the whole RDF(S) triples stored in the system. The graph patterns of the query are set in the `where` clause. The following SPARQL query asks for all resources which have a name and a research interest.

Listing 1.1. A simple SPARQL query

```
SELECT *
WHERE {
    ?x http://tiny/name ?y .
    ?x http://tiny/interest ?z
}
```

The answer of a SPARQL query with a `select` clause is a bag of variable bindings. If the above SPARQL query is posed over the RDF graph of Figure 2, the variable bindings of the answer would be:

?x	?y	?z
`http://tiny/bob`	`"Bob"`	`http://tiny/db`

SPARQL also supports more complex queries than simple BGPs with features such as the `OPTIONAL` operation. The *optional* functionality allows for patterns that might not match for every part in an RDF graph. In this case, the optional part creates no bindings but does not eliminate the entire solution. Keyword `OPTIONAL` is used with a graph pattern such as the ones in the `WHERE` clause.

In addition, one can use filter expressions with operators such as $<$, $=$, and $>$ for numerical values (range constraints) and string functions such as regular expressions. Multiple BGPs can be combined through a `UNION` operation. Order constraints are also possible through `ORDERBY` and `LIMIT` operators.

Apart from the `SELECT` clause, SPARQL also allows `CONSTRUCT, DESCRIBE` and `ASK` clauses. In the case of a `SELECT` query, the result set is a set of variables and their possible bindings. On the other hand, if we have a `CONSTRUCT` query, the result is an RDF graph constructed by substituting variables in a set of triple templates. Finally, a `DESCRIBE` query returns an RDF graph that describes the resources found and an `ASK` query returns yes or no depending on whether a query pattern matches or not. In this chapter, we focus only on `SELECT` queries.

The latest SPARQL 1.1 proposal [31] also supports property paths, negation, aggregates etc. We do not consider these features in the present chapter since most existing works focus on BGP queries. The formal semantics of SPARQL can be found in [63].

3 Single-Source RDF Stores

The storage strategy is probably the defining component of a single-source RDF store, and therefore we structure this section around the storage strategies used by existing stores. Clearly, there are several alternatives to store RDF data, and at a high level we can classify existing strategies into two categories, namely, those using novel *native* representations for RDF data (e.g., Jena TDB [39], RDF-3X [58]), and those using relational databases as the back-end (e.g., Jena SDB [39], DB2RDF [17], C-Store [1]). While there is usually little information available regarding the internals of the former type of stores, the representations used by the latter stores is commonly explained in detail, and this is where we are going to focus our attention in this section.

3.1 "Monolithic" Triple Stores

The simplest and most straightforward representation of RDF data into a relational store is to create a single SPO relation with columns to store the subject, SPO

subj	pred	obj
http://tiny/me	http://tiny/name	“Tasos”
http://tiny/me	http://tiny/attend	“RW 2014”
http://tiny/you	http://tiny/name	“Zoi”
http://tiny/me	http://tiny/friend	http://tiny/you

(a) One version of the “monolithic” store

Dictionary

id	uri-literal
1	http://tiny/me
2	http://tiny/name
3	“Tasos”
4	http://tiny/attend
5	“RW 2014”
6	http://tiny/you
7	“Zoi”
8	http://tiny/friend

SPO

subj	pred	obj
1	2	3
1	4	5
6	2	7
1	8	6

(b) A second version

Fig. 4. The “monolithic” store

predicate and object of each triple. There are multiple ways to *implement* such a store (e.g., see [7]) and Figure 4 shows two alternative versions. In the first version, the URIs and literals are stored directly into the single relation. In the second version, we create a “dictionary” in which we assign an id to each URI or literal. Then, in the *main* relation we store triples created out of these ids, instead of the actual URIs or literals. It is not hard to see that the second version requires less storage space since multiple occurrences of a URI are replaced by a reference to a single id. However, the second version incurs additional costs during query processing since it requires joining the dictionary and SPO relations in order to (re-)establish at query time the relationship between ids and URIs or literals.

Both versions of the store have the desirable characteristic that the schema of the store does not change as we encounter new (types of) triples (in the following sections, we see that this is not a characteristic shared by all the relational representations of RDF). On the negative side, it is clear that the size of the relation storing the triples increases linearly with the number of triples and can very quickly include millions (or billions) of triples/tuples. Managing such a huge relation is bound to have an impact on the evaluation of queries and indeed in “monolithic” stores query performance and scalability do not go hand-in-hand [17]. Therefore, multiple indexes [58] over the single SPO relation are often necessary to support efficient query evaluation.

3.2 Property-Table Stores

The idea behind property tables [83] is the simple observation that entities that are of the same (or similar) type(s) share similar sets of predicates. So, for example, all book entities whose information is represented in RDF are expected to have at least the predicates title, author, and year of copyright. Similarly, all compact-disc entities are expected to also have the predicates of title, artist and year of copyright. Therefore, while designing a schema to store the RDF data for these entities, one can take advantage of these common predicates to create appropriate tables with columns for these common predicates. Figure 5 shows the basic idea through an example. On the left of the figure, we show some sample RDF data pertaining to books, CD’s and DVD’s. Then, in Figure 5(b) we show an *implementation* of the property tables idea, called *property-class tables*, where given a type in the RDF data we create a table with all the common predicates that appear in the instances of this type. So, one table is created in the figure for the BookType entities, and one table is created for the CDType entities. Notice that the *common* predicates among the instances of these types are selected and become the columns of the created tables. Inspired by the monolithic approach, the schema also contains an SPO relation storing the remaining less common predicates for the various entities.

An alternative way to implement property tables is shown in Figure 5(c), through the use of *cluster property tables*. In a nutshell, in this implementation we create a table that contains common predicates between entities, irrespectively of their types. In our example, these are the title and copyright predicates which seem to be present across all BookType, CDType and DVDType entities. Predicates that are not part of the cluster in this example end up in a monolithic SPO table, as it is shown.

Notice that unlike the monolithic schema which is independent of the data being stored, for the creation of property tables we rely heavily on characteristics of the data. However, as the data change through time, so do their characteristics. For example, a predicate that was not as common at some stage in a dataset, it might end up becoming extremely common later on. This would necessitate a change in the schema of the property tables to reflect the fact. However, such data reorganization is usually extremely costly and undesirable. One might consider the monolithic and property table designs as being the two ends of the design spectrum. At one end, we create schemas that closely fit the data and have an associated high cost of periodic reorganization but also hold the promise of efficient query processing. At the other end, we create schemas that are generic and unchanged but are not as efficient in terms querying. There is an interesting challenge here in figuring out whether there are alternative designs where the schema is both unchanged and does not require reorganization, but efficient query processing is not sacrifice. Indeed, in the following sections we show that such designs are possible.

3.3 Vertically Partitioned Stores

The advent of column-oriented DBMS (e.g., see C-store [49], or more recently the BLU feature of IBM DB2 [66]) gave rise to another alternative representation of RDF data, one in which separate tables are created for each predicate appearing in the RDF data. Figure 6 shows the result of vertically partitioning the data of Figure 5(a).

One of the advantages of vertically partitioned stores is efficient query processing. Since SPARQL queries often bind the predicates of triple patterns, during the evaluation of SPARQL queries over vertically partitioned stores the optimizer only accesses the tables corresponding to the predicates mentioned in the query. So, for example if a SPARQL query requires the entities (subjects) that have a copyright predicate, the optimizer only needs to perform a select-* query over the copyright relation of Figure 6. In contrast to this approach, in the

id1	type	BookType
id1	title	“XYZ”
id1	author	“Fox, Joe”
id1	copyright	“2001”
id2	type	CDType
id2	title	“ABC”
id2	artist	“Orr, Tim”
id2	copyright	“1985”
id2	language	“French”
id3	type	BookType
id3	title	“MNO
id3	language	“English”
id4	type	DVDType
id4	title	“DEF”
id5	type	CDType
id5	title	“GHI”
id5	copyright	“1995”
id6	type	BookType
id6	copyright	“2004”

(a) Sample data

BookType

subj	title	author	copyright
id1	“XYZ”	“Fox, Joe”	“2001”
id3	“MNO”	NULL	NULL
id6	NULL	NULL	“2004”

CDType

subj	title	artist	copyright
id2	“ABC”	“Orr, Tim”	“1985”
id5	“GHI”	NULL	“1995”

SPO

subj	pred	obj
id2	language	“French”
id3	language	“English”
id4	type	DVDType
id4	title	“DEF”

(b) Property-class tables

Cluster

subject	type	title	copyright
id1	BookType	“XYZ”	“2001”
id2	CDType	“ABC”	“1985”
id3	BookType	“MNO”	NULL
id4	DVDType	“DEF”	NULL
id5	CDType	“GHI”	“1995”
id6	BookType	NULL	“2004”

SPO

subj	pred	obj
id1	author	“Fox, Joe”
id2	artist	“Orr, Tim”
id2	language	“French”
id3	language	“English”

(c) Cluster property tables

Fig. 5. Property tables

type

subj	obj
id1	BookType
id2	CDType
id3	BookType
id4	DVDType
id5	CDType
id6	BookType

title

subj	obj
id1	“XYZ”
id2	“ABC”
id3	“MNO
id4	“DEF”
id5	“GHI”

copyright

subj	obj
id1	“2001”
id2	“1985”
id5	“1995”
id6	“2004”

language

subj	obj
id2	“French”
id3	“English”

artist

subj	obj
id2	“Orr, Tim”

author

subj	obj
id1	“Fox, Joe”

Fig. 6. Vertically partitioned data

evaluation of the same query over a monolithic store all the triples in the monolithic relation must be accessed, unless an index on the pred column is provided to avoid a sequential scan. On the negative side, the strongest point of the vertically partitioned store is also its weakest. If SPARQL queries do not bind the predicate of triple patterns, then all the relations in the store might need to be accessed to evaluate the query. At the same time, this type of RDF representation suffers from the same shortcoming as the one observed in property tables. As the data evolve over time and, say, a new predicate is added in some entity that was never seen before, the schema of the underlying store must change and a new table must be created. This, coupled with the need to often create indexes for the new tables results in a potentially costly operation.

3.4 Entity-Oriented Stores

The last part of this section covers a novel representation for RDF data that attempts to address the shortcomings of past representations [17]. In more detail, the entity-oriented store uses a representation that similar to the monolithic store does not change over the lifetime of the data. However, unlike the monolithic store and like the property and vertically-partitioned stores, it supports efficient evaluation of SPARQL queries across a wide spectrum of queries.

entity	spill	$pred_1$	val_1	$pred_2$	val_2	$pred_3$	val_3
id1	1	title	"XYZ"	type	BookType	author	"Fox, Joe"
id1	1	copyright	"2001"	NULL	NULL	NULL	NULL
id2	1	title	"ABC"	type	CDType	artist	"Orr, Tim"
id2	1	copyright	"1985"	NULL	NULL	language	"French"
id3	0	title	"MNO	type	BookType	language	"English"
id4	0	title	"DEF"	type	DVDType	NULL	NULL
id5	1	title	"GHI"	type	CDType	NULL	NULL
id5	1	copyright	"1995"	NULL	NULL	NULL	NULL
id6	0	copyright	"2004"	type	BookType	NULL	NULL

Fig. 7. Entity-oriented store

Figure 7 shows the entity-oriented representation for the data shown in Figure 5(a). The first thing to observe in this representation is that unlike any of the previous representations the columns of the relation has no pre-assigned semantics. That is, the table in the particular example has 3 pairs of ($pred_i$, val_i) without specifying which predicate-object pairs are to be stored there. Indeed, the assignment of predicate-object pairs to ($pred_i$, val_i) columns happens dynamically by the system, during data loading, and is one of the distinguishing characteristics of the representation. During this assignment, a predicate will always be *hashed* to the same column. So, predicate title is always hashed to column $pred_1$ for any entity having this predicate, while predicate artist is always hashed to column $pred_3$. However, notice that multiple predicates can be hashed to the same column. For example, predicate author is also hashed to $pred_3$. The entity oriented store provides a host of alternative strategies so as to optimize this assignment of predicates to columns with the main objective to avoid *collisions* (i.e., multiple predicates being hashed to the same column for the same entity) while at the same time maximizing compression (i.e., allowing multiple predicates to be hashed to the same column when these predicates are not co-occurring in entities). The end result is a representation that has both a small footprint in terms of space requirements and offers superior query performance when compared with other representations (for more details see [17]).

4 P2P-Based RDF Stores

Peer-to-peer (P2P) networks had initially emerged as a natural way for file sharing in a decentralized manner. Popular systems such as Napster[1], Gnutella[2], Freenet[3], Kazaa[4], Morpheus[5] had made this model of interaction popular. In P2P systems a very large number of autonomous computing nodes (the *peers*) pool together their resources and rely on each other for *data* and *services*. In contrast with a client-server architecture, P2P nodes serve as both a provider

[1] `http://www.napster.com`

[2] There are various clients implementing the Gnutella protocol or variations. See for example, `http://www.limewire.com`.

[3] `http://freenet.sourceforge.net`

[4] `http://www.kazaa.com`

[5] `http://www.musiccity.com`

and a consumer of resources. P2P networks are typically distinguished into three different classes according to their topology: *unstructured*, *structured* and *hierarchical* networks.

In unstructured networks all peers are equal and form an overlay network with no restrictions on topology and no centralized source of information. Such systems are highly resilient to churn, i.e., when peers join and leave the network. However, they flood the network with messages to find a piece of data and cannot guarantee to find it in a reasonable amount of hops. Gnutella and Kazaa form examples of such networks. On the contrary, structured networks have a regular topology, e.g., rings or hypercubes, and were devised as a remedy for the routing and object location inefficiencies of unstructured networks. A very popular class of structured networks is the distributed hash tables (DHTs). Hierarchical networks partition the nodes into two categories: super-peers and clients. In such a network, all super-peers are equal and have the same responsibilities: serving a fraction of the clients and keeping indices of the resources of those clients. Super-peers interact by following a protocol of their choice (e.g., a symmetric one like Gnutella, a structured one like Napster or a DHT protocol). Clients can run on user computers and are equal to each other running the same software. Clients learn about resources by querying super-peers and downloading resources directly from other clients.

The combination of Semantic Web technologies (i.e., RDF, RDFS and ontologies) and P2P systems provided accurate data retrieval and efficient search in distributed application scenarios, thus, it has been the focus of many research works the past few years. In the following, we present representative works in this research area where peers are organized in a *structured overlay network* or a *schema-based network*. We mainly focus on how query processing is performed in such distributed settings. The local data access at each peer can be achieved by the various techniques proposed in the previous section. A more comprehensive survey of on P2P-based RDF management can be found in [79].

4.1 Structured Overlay Networks

Distributed hash tables (DHTs) is a prominent class of structured overlays that attempt to solve the object *lookup problem*: given a data item x, find the node which holds x. Each node and each data item is assigned a unique m-bit identifier by using a hashing function such as SHA-1 [76]. The identifier of a node can be computed by hashing its IP address. For data items, we first have to compute a *key* and then hash this key to obtain an identifier *id*. The node with identifier that is numerically closest to *id* is responsible for storing the data item x. The lookup problem is then solved in $O(log(n))$ hops in a network of n peers by providing a simple interface of two requests: $\textsc{Put}(id, x)$ and $\textsc{Get}(id)$.

We focus on representative RDF stores that use an underlying DHT to store and query RDF data. These include RDFPeers [22], Atlas [40], BabelPeers [15], UniStore [43], RDFCube [53] and GridVine [3].

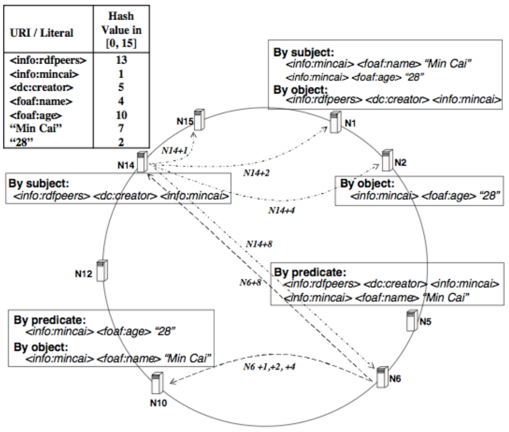
URI / Literal
Hash Value in [0, 15]
<info:rdfpeers> 13
<info:mincai> 1
<dc:creator> 5
<foaf:name> 4
<foaf:age> 10
"Min Cai" 7
"28" 2
By subject: <info:mincai> <foaf:name> "Min Cai" <info:mincai> <foaf:age> "28"
By object: <info:rdfpeers> <dc:creator> <info:mincai>
By subject: <info:rdfpeers> <dc:creator> <info:mincai>
By object: <info:mincai> <foaf:age> "28"
By predicate: <info:rdfpeers> <dc:creator> <info:mincai> <info:mincai> <foaf:name> "Min Cai"
By predicate: <info:mincai> <foaf:age> "28"
By object: <info:mincai> <foaf:name> "Min Cai"
N1
N2
N5
N6
N10
N12
N14
N15
N14+1
N14+2
N14+4
N14+8
N6+8
N6 +1,+2,+4

Fig. 8. RDFPeers data partitioning example from [20]

Data Partitioning. In the distributed environment of a DHT, one has to decide how to partition the RDF data. A commonly used data partitioning scheme in DHT-based systems is achieved through the hashing of some or all RDF elements of the triples. RDFPeers [20, 22] is the first system that came up with this partitioning scheme and has influenced significantly many follow-up works. RDFPeers is implemented on top of MAAN [21], a self-organized DHT network which extends the well-known DHT protocol of Chord [81] to efficiently answer multi-attribute and range queries.

In RDFPeers, each node uses the RDF data model to create descriptions of resources that it wants to make available to the rest of the network nodes. Each RDF triple in RDF document is indexed to three different network nodes: it is stored once in the node responsible for the identifier that is computed by hashing the subject value of the triple, and twice more by using the predicate and object values of the triple. The SHA-1 hash function [76] is used if the value is a string. If the value is a numeric one then an order preserving hash function is used, which allows efficient evaluation of range queries. Figure 8 shows the indexing of some triples in a network of 8 nodes with a 4-bit identifier space.

GridVine uses P-Grid [2], a structured overlay network based on the principles of DHTs with lexicographic key ordering. Peers in GridVine are able to publish available resources by creating RDF triples (metadata). An RDF triple is stored three times in the network using three different keys based on its subject, predicate and object values, as in [22]. In addition, prefix indexing, e.g., on the beginning of a string representing an object value, can be easily supported using P-Grid routing mechanisms.

BabelPeers [15, 33, 34] and Atlas [40, 41], two more recent systems built on top of Pastry [70] and BambooDHT [68] respectively, also use the triple indexing scheme, storing each triple three times in the network once for each RDF element, as initially proposed in [22]. Unistore [43, 44] can index each triple multiple times, using different (combinations of) triple elements in P-Grid [2].

Metadata and Source Selection. In structured overlay networks, and specifically in DHTs, there is no need to keep metadata about identifying which piece of data is located in which node. This is implicitly done by the distributed index they provide for the lookup problem. The problem of source selection is then tackled easily and efficiently by the use of hash functions: the hash value of a key is used to retrieve the node that keeps the data item with this key. DHT-based RDF stores use the constant part(s) of triple patterns involved in the query to compute the identifiers that lead to the nodes storing matching triples.

Query Planning and Evaluation. After the peers that contain matching triples for a triple pattern have been identified, a single triple pattern query can be easily answered by contacting this peer and retrieving the triples that match the triple pattern. Single triple pattern queries require $O(log(n))$ routing hops in a network of n peers, except for the triple pattern without any constant, i.e., `(?x, ?y, ?z)`, where all n peers need to be contacted.

RDFPeers [20] support single triple pattern queries, disjunctive and range queries and conjunctive multi-predicate queries, i.e., conjunctive queries with the join variable on the subject value and the predicate constant. Query execution is performed sequentially at the peers that contain matching triples for the triple patterns of the query and the results are returned to the peer that posed the query. The algorithm finds candidate subjects on each triple pattern recursively and intersects the candidate subjects found at the peer with the found candidate sets for the previously evaluated triple patterns, before returning the search results to the query requestor.

In Atlas [40] the query processing algorithm, QC, works sequentially at the peers containing matching triples, extending the one proposed in [20] for any kind of conjunctive query. With QC the query is evaluated by a chain of nodes. Intermediate results flow through the nodes of this chain and finally the last node in the chain delivers the result back to the node that submitted the query. In [52], an additional query processing algorithm, SBV, is described where the values found to match a triple pattern are used to rewrite the following patterns. SBV achieves a better distribution of the query processing load. It does not create a single chain for a query as QC, but by exploiting the values of matching triples found while processing the query incrementally, it rewrites the query and distributes the responsibility of evaluating it to more nodes than QC (in other words, SBV is constructing multiple chains for each query). Obviously, the order in which the triple patterns are evaluated affects the query performance and in [41] some query optimization techniques are proposed to achieve better query response times.

In BabelPeers [15, 33, 34], query planning and evaluation is performed at a single peer, the one that receives the query request, and includes two phases. In the first phase, all candidate sets for all triples and variables are retrieved from the various nodes of the network to the node that receives the query request. The second phase of the query evaluation, includes local processing of the candidate sets to find the actual answer to the query. The disadvantage of this approach is that the node that receives the query request has to do all the computation and suffers a lot of query processing load. In addition, the candidate sets transferred through the network might contain results that will never be used in the final answer of the query. This causes unnecessary traffic to the network. The authors of [33,34] propose some methods to remove the amount of the useless information from the candidate sets using Bloom filters. Finally, [14] addresses the problem of uneven load among the nodes of BabelPeers due to the skewness of the RDF datasets and proposes several techniques for load balancing.

GridVine [3] follows the principles of data independency and separates the logical from the physical layer. The logical layer consists of the semantic overlay for managing and mapping data and metadata schemas, while the physical layer consists of a structured P2P overlay network that efficiently routes messages. The latter is used to implement various functions at the logical layer, like attribute-based search, schema management and schema mapping management. GridVine allows peers to derive new schemas from well-known base schemas (using RDFS), providing schema inheritance. Each peer has also the possibility to create a mapping between two schemas, in which case translation links among network peers are created (using OWL). In this way, queries are propagated from one semantic domain to another. There are two approaches used for resolving translation links, the iterative and the recursive resolution. With iterative resolution, the peer issuing an RDF query tries to find and process all translation links by itself, while with recursive resolution more than one peers are involved by delegating the query and its translations.

In Unistore [43,44] each query is transformed into a logical query plan which is in turn transformed into a physical query plan using operators defined in [43]. Query planning is performed dynamically at each peer involved in the query evaluation. Unistore uses a cost-based optimizer which estimates the cost of physical operators in terms of the number of hops and messages required for each operator.

4.2 Schema-Based P2P Architectures

In contrast to the RDF stores based on a structured overlay architecture, where data is partitioned among the peers in a specific way and a distributed index is used to locate data, in schema-based P2P systems peers keep the data they have locally and they work collaboratively by exploiting their schemas to tackle the query processing problem. Two influential schema-based P2P for storing and querying RDF data are Edutella [55–57] and SQPeer [46].

Edutella [55–57] is built on top of a super-peer topology, where there are two kind of peers: the super-peers and the clients. The super-peers are organized under the HyperCup topology [74], while clients are connected to super-peers in a star-like fashion. Each client connects to one super-peer only, while a super-peer can have multiple clients and is used to efficiently handle all the requests of clients.

SQPeer [46,47,78] is a middleware for routing and planning complex queries in P2P database systems, exploiting the schemas of peers. SQPeer has two different architecture alternatives: it can be organized in a super-peer network or in DHT-based one. Peers that employ the same RDFS schema belong to the same semantic overlay network (SON) [82] and queries posed by peers should conform to the RDFS schema of the SON they belong.

Metadata and Source Selection. In Edutella [26], when a peer joins the network, it provides its super-peer with its metadata information, i.e., a description of the metadata that has been created by this client (supported schema, used values etc.). The actual metadata remains in the client peer. Each super-peer employs two routing indices: The super-peer/peer (SP/P) indices that contain information about each peer that is connected to the super-peer, and the super-peer/super-peer (SP/SP) indices that are extracted summaries from all local SP/P indices. The SP/P indices keep information about peers at different granularities: schemas, schema properties, property value ranges and individual property values. Both types of indices also contain statistics for optimization purposes (size of documents, network characteristics, etc.). This information is used to efficiently route queries only to super-peers and clients that may contain an answer to the query.

In SQPeer, each peer provides the RDFS about the resources that wants to make available in the network. Peers that employ the same schema belong essentially to the same semantic overlay network (SON) [82]. Each peer *advertises* the content (the data values or the schema) of its local database. The RDF Schema defining a SON may contain numerous classes and properties not necessarily populated in a peer's database. For this reason, a peer uses virtual or materialized views to specify the fragment of the schema for which all classes and properties are (in the materialized scenario) or can be (in the virtual scenario) described in a peer's local database. These views may be broadcast to (or requested by) other peers, thus informing the rest of the P2P system of the information available in the peers' databases.

The view propagation in SQPeer depends on the underlying architecture (super-peer or DHT). In the super-peer architecture, a peer that connects to a super-peer forwards its corresponding view and all super-peers are aware of each other. This enables the processing of queries expressed in terms of different RDFS schemata (or fragments). Source selection in a DHT-based SQPeer is done as follows. Unique keys are assigned to each view pattern and hence peers, whose hash values match those keys, are aware of the peer bases that are populated with data answering of a specific schema fragment. An appropriate key assignment and hash function is used for neighbor peers to hold successive view patterns with respect to the class/property hierarchy defined in the employed RDF Schema.

Query Planning and Evaluation. Query planning in Edutella is performed dynamically and not at a single site because super-peers have a very limited view of the whole P2P network (only the neighbors are known), and thus, no comprehensive static plan in the traditional sense can be produced. The query plan chosen by the optimizer is split into a local plan and multiple remote query plans. The remote plans are shipped to the referenced hosts where the optimization process continues on the smaller query plans. The local query plan is instantiated and combines the results of the remote query plans.

Queries in SQPeer are formulated according to the RDF schema that the requester peer supports. The proposed query routing and query processing algorithm can find the relevant peers that actually answer each query and generate query plans by taking into account statistics on data distributions. SQPeer supports two kinds of query optimization for query planning, i.e., compile-time and run-time optimization. The former uses heuristics and statistics to push as much as possible processing to the same peers and decide at compile time among data, query or hybrid shipping execution strategies. On the other hand, run-time optimization includes deciding at execution time on altering the data or query shipping decision or discovering alternative peers for answering a certain part of a query plan.

5 Federated RDF Stores

Probably one of the most distinguishing characteristics of a federated RDF store is that the system is built in a *bottom-up* fashion. That is, existing pre-populated stores come together in order to provide a way to evaluate queries across all of them. Therefore, in federated systems data partitioning is essentially a non-issue and the system has no control as to where data will reside.

Although, there is a considerable number of systems in this category, we are going to limit our presentation to only a subset of them and try to highlight the main design choices through them. So, in the following paragraphs we focus our presentation to a handful of systems including FedX [75], SPLENDID [28] and ANAPSID [4]. For a more extended and in-depth description of the systems, including an evaluation of their performance the reader is encourage to consider related surveys in this space [65, 72].

Metadata and Source Selection. Federated systems have no control or *a priori* knowledge as to where data reside. Still, this information is critical for the evaluation of queries since the system must be able to locate the data that is relevant to the query at hand. In a nutshell, given a query there are two alternative approaches in acquiring the necessary metadata and guiding source selection. The first approach is to determine the metadata at *query time*. This is mostly done by forming appropriate *polling* queries, usually in the form of SPARQL ASK (boolean) queries and using the results of these queries to determine sources that are relevant to a query. The clear benefit of this approach is that no additional metadata are maintained by the federated system (which may need to be updated/maintained as the data in the underlying source change). On the negative side, end-to-end query evaluation time might increase since in addition to the actual query, metadata computation happens at query time and the cost of evaluating ASK queries is not negligible. FedX [75] is an example of a system that maintains no metadata and uses query-time determination of sources (though the system does provide an option for caching to avoid some of the query time costs).

The second approach relies on actually maintaining in the federation some form of description for the contents of the underlying sources. Such metadata can take the form of voiD [6] descriptions used by SPLENDID [28], and can include from high level information like the location of a dataset, to low level information like statistics about the number of triples in a dataset or the number of instances of a property in the data. Other metadata representations, like the one used by ANAPSID [4], are inspired by work in relational databases and employ Local-as-View (LAV) definitions [30] to describe the data stores in individual sources. Yet another alternative includes building summarized indexes [51] describing the contents of the underlying RDF sources, and using such indexes to guide query evaluation [35].

What metadata is available to each system, and how well the system takes advantage of the available metadata are both factors that influence the effectiveness of a source selection strategy. In more detail, using the available metadata, the objective of a source selection strategy is to be both *sound* and *complete*. In this context, completeness is used to denote the desirable property that a source selection strategy identifies *all* the sources that contain data relevant to the query (and thus does not miss any query results). Unfortunately, not all systems are complete with FedX being probably one of the few popular systems having this property [72]. In terms of soundness, the term denotes the desirable property that a source selection strategy should ideally involve *only* the sources that contain data relevant to the query, and no other *irrelevant* sources. Soundness is a property that is closely tied to performance. If a system is not sound then it is bound to contact a lot of irrelevant sources, which results in unnecessary work and prolongs query evaluation time. Not surprisingly, it is particularly hard to achieve soundness in a federated environment and indeed all of the systems do some amount of unnecessary work. Therefore, in the context of soundness, the objective in most systems is minimizing this amount of work, ideally without sacrificing completeness (see [72] for a study of how well existing systems perform in terms of soundness).

Query Planning and Evaluation. Similarly to the centralized setting where once we identify the relations that participate in a query we have to decide the order with which the relations will be joined, in the federated setting once the sources to be involved in the query evaluation are selected, the order with which these sources will be processed needs to be determined. One of the main objectives while deciding the proper ordering is to minimize the intermediate results computed as the results from different sources are correlated (joined). The available metadata, as well as statistics from the evaluated ASK queries during source selection are commonly used to decide this ordering. As the ordering is determined, each system must also decide the join strategy to be used. Bind joins [29], hash joins and nested loop joins are typical strategies used across a variety of systems, with many systems offering more than one strategy. For example, while FedX supports bind joins, SPLENDID supports both hash and bind joins.

As the last step, the actual evaluation of the query takes place. There are still several challenges to be addressed even at this point in the process. For example, a single input query might *spawn* multiple sub-queries over the federated data. Some of these sub-queries might retrieve the same result and therefore one of the challenges is whether the federated system can actually detect these duplicate answers. As another example, being by nature a distributed system, the federation should be able to handle source failures. So, if in the middle of a query evaluation one of the sources leaves the federation (by choice or due to a crash), the system should be able to detect this failure and inform the remaining sources that participate in the federated query.

6 Cloud-Based RDF Stores

The recent explosion in the size of data that is generated and used in various applications, also termed as *big data*, has led to the emergence of new technologies that (i) can scale to large amounts of data, (ii) provide fault-tolerance, (iii) allow for elastic allocation of machines and (iv) free the developer/user from the burden of hardware and software administration. These technologies enable the easy deployment of distributed architectures and are often termed as *cloud computing*. Example systems offering such features, either as a service in the cloud or in a private cluster, are the reputed *NoSQL key-value stores* [23]. At the same time, interest in massively parallel processing has been renewed by the *MapReduce* proposal [25] and many follow-up works, which aim at solving large-volume data management tasks based in a cloud environment, or more generally-speaking in a large-scale distributed platform. We first briefly describe the functionalities offered by MapReduce and distributed key-value stores.

MapReduce. Interest in massively parallel processing has been renewed recently since the emergence of the MapReduce framework [25] and its open source implementation Hadoop [10]. MapReduce has become popular in various computer-science fields as it provides a simple programming paradigm which frees the developer from the burden of handling parallelization, scalability, load balancing and fault-tolerance. MapReduce processing is organized in *jobs*. Each job consists of a *map* and a *reduce* phase, separated by a *shuffle* (data transfer) phase. The map phase is specified by a user-defined function which takes as input (key, value) pairs, performs some tasks on these (if needed) and outputs intermediate pairs (ikey, ivalue). These pairs are shuffled through the network and are given as input to the reduce phase. The distributed file system (DFS) of Hadoop, HDFS, splits data into blocks and each map task operates on a separate block of data. The nodes of the cluster run in parallel one or more map/reduce tasks. A comprehensive survey on MapReduce and its extensions can be found in [27, 71].

Key-value stores. Distributed key-value stores provide very simple data structures based on the concept of $(key, value)$ pairs. Such stores typically handle items, each of which consist of a key and several attributes; in turn, an attribute consists of a name and one or several values. For convenience, most key-value stores also support named collections of items, which are typically called tables. An overview of various key-value stores can be found in [23].

In the rest of the section we give an overview of the most recent advances in cloud-based RDF stores with a focus on systems that store data in the MapReduce file system, in NoSQL key-value stores, in multiple centralized RDF stores or in a commercial cloud. A more detailed survey in this research area can be found in [42].

6.1 RDF Stores on MapReduce and DFS

This category includes works that use MapReduce and its underlying distributed file system. These systems are built to make the most out of the parallel processing capacities provided by the underlying MapReduce paradigm. RDF data is stored in files which are split by the distributed file system in the cluster nodes. Their negative aspect from the perspective of the data store is that they do not have efficient fine-grained data stores to rely on. Representative works include SHARD [69], HadoopRDF [38], RAPID+ [45,67] and PigSPARQL [73]. *Metadata* collection and *source selection* in these systems is handled by the MapReduce framework (the namenode and scheduler) and thus, is a non-issue.

Data Partitioning. In these systems the user specifies how the data is partitioned at the file level: the user/system uploads to the system files which contain the RDF triples. There are two commonly used ways to store the data in a file system. In the first one, the data stored in the files are stored based on the triple model, i.e., one triple per line in each file. The second approach contains works that organize the data in predicate-based files. Each predicate file contains the subjects and objects of triples with a specific predicate. Conceptually the first approach resembles the monolithic approach used in single-source approaches, while the latter resembles the vertical partitioning where one relation is created per predicate (see Section 3). As MapReduce does not provide fine-grained indices, the latter way is preferred because it decreases the amount of data that need to be scanned during query evaluation. [69,73] belong to the first category, while [38,67,86] belong to the second one. Note that the placement of the data blocks is performed by the underlying distributed file system.

A different approach is followed in EAGRE [85] where the goal is to reduce the I/O cost during query processing. First, the RDF graph is compressed to an entity-based graph where entities (subjects of triples) with similar properties are grouped in class entities. Then, the compressed graph is partitioned with METIS [54], a graph partitioning tool, into equal to the size of the cluster partitions. Triples are placed in the partition their entity class belongs to.

Query Planning and Evaluation. Usually query planning takes place at a client node and concerns the query decomposition of a query into subqueries to be processed in parallel. The query decomposition can lead to either left-deep or bushy query plans being built. Bushy query plans are better suited for parallel query evaluation as they can exploit both intra- and inter-operator parallelism. However, their search space is very large compared to the one of left-deep trees and for this reason most works build simple left-deep trees, such as SHARD [69] and PigSPARQL [73], or use a hybrid approach where the leaves of the query plan is in bushy shape while the intermediate results are processed in a left-deep manner such as RAPID+ [67]. The only work that builds fully bushy plans is HadoopRDF [38] using a heuristic approach to minimize the number of MapReduce jobs for pruning the search space.

Then, query evaluation is performed using MapReduce jobs. For left-deep query plans one job is performed per join. Since MapReduce does not provide a join functionality, the system has to implement its own join operators. There is wide literature on how to implement a join in MapReduce with the repartition and broadcast join to be the most common ones. The interested reader may refer to [5,16].

6.2 RDF Stores on Top of NoSQL Key-Value Stores

There are many works that use NoSQL key-value stores as back-ends for storing and indexing RDF data. These systems benefit from the efficient and fine-grained storage and retrieval of the key-value stores, however, they suffer in more complex functionalities such as joins. Representatives of the second category include systems such as Rya [64] which uses Apache Accumulo [8], Trinity.RDF [84] which is built on top of a in-memory key-value store [77], CumulusRDF [48] based on Apache Cassandra [9], and H_2RDF+ [61] built on top of HBase [11]. Similarly with the previous set of works on MapReduce, *metadata* collection and *source selection* is performed by the key-value store opaquely to the user and we thus, omit any discussion about them.

Data Partitioning. Data partitioning in key-value stores amounts to the choice of indices that will be build by each system. As shown in Section 3, centralized RDF stores usually use extensive indexing schemes that enable fast data access for all triple patterns and efficient performing of merge-joins. This extensive indexing scheme has a significant storage overhead which is amplified in a cloud environment where data is also replicated for fault-tolerance reasons. For this reason, most RDF stores built on top of key-value stores employ a subset of these indices which is sufficient for matching efficiently all possible triple patterns. The three permutations massively used are subject-predicate-object (SPO), predicate-object-subject (POS) and object-subject-predicate (OSP). Typically systems materialize each of one of these indices in a separate table in the key-value store. The choice of the keys and values highly depends on the underlying functionality of the key-value store, e.g., if it provides a range scan over the key space. For instance, each of the subject, predicate, object can be mapped to a key in the key value store, or a concatenation of two or three of the RDF elements can be used as key if a range scan is supported by the key-value store.

Query Planning and Evaluation. As key-value stores do not allow for performing joins on the server-side, query planning and query evaluation is often performed at a single site. This is simply done by fetching the triples from the key-value store that match the individual triple patterns and performing the join locally at the client-side. This approach is followed by most systems, e.g., [48,64]. A hybrid approach is proposed in H_2RDF+ [61] where either a centralized local join is performed at the client side or a MapReduce job is instantiated depending on the selectivity of the join. Trinity.RDF [84] uses a graph-based approach by navigating the RDF graph and finding matches to the query.

6.3 Approaches Using Multiple Centralized RDF Stores

Within the third category, centralized RDF stores distributed among multiple nodes are used to exploit parallelization such as in [36, 37]. These systems are based on a master/slave architecture, where the master partitions and places the RDF triples in the slave nodes. Each slave node stores its local RDF triples in a centralized RDF store such as RDF-3X.

Data Partitioning. The goal is a data partitioning scheme that enables high parallelization during query evaluation while striving to minimize communication among the slave nodes. In [37] a graph partitioning tool, called METIS [54], is used to partition the RDF graph into as many partitions as the number of nodes so that a minimum number of edges is cut, i.e., the minimum number of triples have their subject and object in different partitions. Placement is done by assigning each triple in the partition its subject belongs to, termed as 1-hop directed guarantee, or to the partitions that both the subject and object belongs to, termed as 1-hop undirected guarantee. This leads to replicating the triples that are on the edge cuts. There is also the possibility to allow for further replication of those triples that are at partition boundaries. A directed (undirected) n-hop guarantee is achieved when any triples forming a directed (undirected) path of length n will be located within the same partition. A similar approach is followed in [36] with the difference that replication occurs only in the parts of data that are certain to be accessed for a given query workload.

Metadata and Source Selection. Although data is partitioned following a specific scheme, in [36, 37] there is no metadata information for mapping the triples to the nodes they are stored. Therefore, a subquery is sent to all the sources to be answered.

Query Planning and Evaluation. Query planning and execution is performed by decomposing the query to subqueries that can be completely answered by the underlying RDF store. If a query can be completely answered by the underlying RDF store then no communication is required. The answers of the query is the union of the individual results. In any other case, network communication is necessary to join the intermediate results of the query which is done in the MapReduce framework.

6.4 RDF Stores in Commercial Clouds

The first store that proposed an RDF store built in a commercial cloud is Stratustore [80] which relies on Amazons SimpleDB [13], an early developed key-value store of Amazon Web Services. In Stratustore, RDF data is indexed in the key-value store and query processing is performed at the client side running at an EC2 machine.

In AMADA [12, 19], a cloud-resident RDF store, a different approach is followed. Raw RDF data reside in Amazon's storage service (S3) as simple files and a file index built in Amazon's key-value store keeps *metadata* on which data can be found in which files. *Source selection* is performed by consulting the indices built to retrieve the files that contain triples that match the query. During *query evaluation* the triples contained in the files selected by the index are cached in a centralized RDF store where query answering is performed.

7 Conclusions

We have presented an overview of query processing techniques for RDF databases. We identified a set of tasks that are of great importance for RDF query processing and that a system should address. These include metadata collection, source selection, data partitioning and query planning and execution in a decentralized environment and storage/indexing and query planning and evaluation in a centralized system. We analyzed each one of these tasks in different architectures. We first discussed issues related to single-source RDF stores and then navigated through P2P systems, federated architectures and cloud-based proposals.

There are numerous of open problems that are yet to be solved. These range from more advanced techniques for query optimization (query decomposition, join ordering, etc.), to building more sophisticated indices or materialized views for speeding up query performance. An important aspect that is usually neglected in the works we have presented in this chapter is RDFS reasoning. RDFS reasoning is an essential functionality of the RDF model and should, therefore, be taken into account when building systems for answering SPARQL queries.

References

1. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB, pp. 411–422 (2007)

2. Aberer, K., Cudre-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-Grid: A Self-Organizing Structured P2P System. SIGMOD Record 32, 29–33 (2003)
3. Aberer, K., Cudre-Mauroux, P., Hauswirth, M., Pelt, T.V.: GridVine: Building Internet-Scale Semantic Overlay Networks. In: Proceedings of the 13th World Wide Web Conference (WWW 2004), New York, USA (2004)
4. Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J., Ruckhaus, E.: Anapsid: An adaptive query processing engine for sparql endpoints. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 18–34. Springer, Heidelberg (2011)
5. Afrati, F.N., Ullman, J.D.: Optimizing Multiway Joins in a Map-Reduce Environment. IEEE Trans. Knowl. Data Eng. 23(9) (2011)
6. Alexander, K., Hausenblas, M.: Describing linked datasets - on the design and usage of void, the vocabulary of interlinked datasets. In: Linked Data on the Web Workshop (LDOW 09), in conjunction with 18th International World Wide Web Conference, WWW 2009 (2009)
7. Alexander, N., Lopez, X., Ravada, S., Stephens, S., Wang, J.: Rdf data model in oracle
8. Apache Accumulo (2012), `http://accumulo.apache.org/`
9. Apache Cassandra (2012), `http://cassandra.apache.org/`
10. Apache Hadoop (2012), `http://hadoop.apache.org/`
11. Apache HBase (2012), `http://hbase.apache.org/`
12. Aranda-Andújar, A., Bugiotti, F., Camacho-Rodríguez, J., Colazzo, D., Goasdoué, F., Kaoudi, Z., Manolescu, I.: Amada: Web Data Repositories in the Amazon Cloud (demo). In: CIKM (2012)
13. Amazon Web Services (2012), `http://aws.amazon.com/`
14. Battre, D., Heine, F., Hoing, A., Kao, O.: Load-balancing in P2P based RDF stores. In: Proceedings of the 2nd International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006, Co-located with ISWC 2006), Athens, Georgia, USA (2006)
15. Battre, D., Heine, F., Hoing, A., Kao, O.: BabelPeers: P2P based Semantic Grid Resource Discovery. High Performance Computing and Grids in Action 16, 288–307 (2008)
16. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A Comparison of Join Algorithms for Log Processing in MapReduce. In: SIGMOD (2010)
17. Bornea, M.A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., Bhattacharjee, B.: Building an efficient RDF store over a relational database. In: SIGMOD Conference, pp. 121–132 (2013)
18. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, W3C Recommendation (2004)
19. Bugiotti, F., Goasdoué, F., Kaoudi, Z., Manolescu, I.: RDF Data Management in the Amazon Cloud. In: DanaC Workshop (in Conjunction with EDBT) (2012)
20. Cai, M., Frank, M.: RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. In: Proceedings of the 13th World Wide Web Conference (WWW 2004), New York, USA (2004)
21. Cai, M., Frank, M., Szekely, P.: MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In: Proceedings of the 4th International Workshop on Grid Computing (Grid2003), Phoenix, Arizona, USA (2003)
22. Cai, M., Frank, M.R., Yan, B., MacGregor, R.M.: A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. Journal of Web Semantics: Science, Services and Agents on the World Wide Web 2(2), 109–130 (2004)

23. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Record 39(4), 12–27 (2011)
24. Chaudhry, N.A., Shaw, K., Abdelguerfi, M. (eds.): Stream Data Management. Advances in Database Systems, vol. 30. Springer (2005)
25. Dean, J., Ghemawat, S.: Mapreduce: Simplified Data Processing on Large Clusters. In: Proceedings of the USENIX Symposium on Operating Systems Design & Implementation (OSDI), pp. 137–147 (2004)
26. Dhraief, H., Kemper, A., Nejdl, W., Wiesner, C.: Processing and Optimization of Complex Queries in Schema-Based P2P-Networks. In: Ng, W.S., Ooi, B.-C., Ouksel, A.M., Sartori, C. (eds.) DBISP2P 2004. LNCS, vol. 3367, pp. 31–45. Springer, Heidelberg (2005)
27. Doulkeridis, C., Norvag, K.: A survey of large-scale analytical query processing in MapReduce. VLDB Journal (2013)
28. Görlitz, O., Staab, S.: Splendid: Sparql endpoint federation exploiting void descriptions. In: COLD (2011)
29. Haas, L.M., Kossmann, D., Wimmers, E.L., Yang, J.: Optimizing queries across diverse data sources. In: Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB 1997, pp. 276–285 (1997)
30. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal 10(4), 270–294 (2001)
31. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation (2013), `http://www.w3.org/TR/sparql11-overview/`
32. Hayes, P.: RDF Semantics. W3C Recommendation (February 2004), `http://www.w3.org/TR/rdf-mt/`
33. Heine, F.: Scalable P2P based RDF Querying. In: Proceedings of the 1st International Conference on Scalable Information Systems (Infoscale 2006), Hong Kong (2006)
34. Heine, F., Hovestadt, M., Kao, O.: Processing Complex RDF Queries over P2P Networks. In: Proceedings of Workshop on Information Retrieval in Peer-to-Peer-Networks (P2PIR 2005), Bremen, Germany (2005)
35. Hoffmann, J., Selman, B. (eds.): Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, Toronto, Ontario, Canada, July 22-26. AAAI Press (2012)
36. Hose, K., Schenkel, R.: WARP: Workload-Aware Replication and Partitioning for RDF. In: DESWEB Workshop (in Conjunction with ICDE) (2013)
37. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4(11), 1123–1134 (2011)
38. Husain, M., McGlothlin, J., Masud, M.M., Khan, L., Thuraisingham, B.M.: Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. IEEE Trans. on Knowl. and Data Eng. (2011)
39. Jena: a semantic web framework for java, `https://jena.apache.org`
40. Kaoudi, Z., Koubarakis, M., Kyzirakos, K., Miliaraki, I., Magiridou, M., Papadakis-Pesaresi, A.: Atlas: Storing, Updating and Querying RDF(S) Data on Top of DHTs. Journal of Web Semantics (2010)
41. Kaoudi, Z., Kyzirakos, K., Koubarakis, M.: SPARQL Query Optimization on Top of DHTs. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 418–435. Springer, Heidelberg (2010)
42. Kaoudi, Z., Manolescu, I.: RDF in the Clouds: A Survey. The VLDB Journal (2014)
43. Karnstedt, M.: Query Processing in a DHT-Based Universal Storage - The World as a Peer-to-Peer Database. PhD thesis (2009)

44. Karnstedt, M., Sattler, K.-U., Richtarsky, M., Muller, J., Hauswirth, M., Schmidt, R., John, R.: UniStore: Querying a DHT-based Universal Storage. In: Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007 (Demo paper), Istanbul, Turkey (April 2007)
45. Kim, H., Ravindra, P., Anyanwu, K.: From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra (demo). PVLDB 4(12), 1426–1429 (2011)
46. Kokkinidis, G., Christophides, V.: Semantic Query Routing and Processing in P2P Database Systems: The ICS-FORTH SQPeer Middleware. In: EDBT Workshops, Heraklion, Crete, Greece (March 2004)
47. Kokkinidis, G., Sidirourgos, L., Christophides, V.: Query Processing in RDF/S-based P2P Database Systems. In: Semantic Web and Peer-to-Peer. Springer (2006)
48. Ladwig, G., Harth, A.: CumulusRDF: Linked Data Management on Nested Key-Value Stores. In: SSWS (2011)
49. Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., Bear, C.: The Vertica Analytic Database: C-store 7 Years Later. In: Proc. VLDB Endow., vol. 5(12), pp. 1790–1801 (2012)
50. Le, W., Kementsietsidis, A., Duan, S., Li, F.: Scalable multi-query optimization for sparql. In: ICDE, pp. 666–677 (2012)
51. Li, F., Le, W., Duan, S., Kementsietsidis, A.: Scalable Keyword Search on Large RDF Data. IEEE Transactions on Knowledge and Data Engineering 99(PrePrints) (2014)
52. Liarou, E., Idreos, S., Koubarakis, M.: Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 399–413. Springer, Heidelberg (2006)
53. Matono, A., Pahlevi, S.M., Kojima, I.: RDFCube: A P2P-Based Three-Dimensional Index for Structural Joins on Distributed Triple Stores. In: Moro, G., Bergamaschi, S., Joseph, S., Morin, J.-H., Ouksel, A.M. (eds.) DBISP2P 2005/2006. LNCS, vol. 4125, pp. 323–330. Springer, Heidelberg (2007)
54. METIS, `http://glaros.dtc.umn.edu/gkhome/views/metis`
55. Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmér, M., Risch, T.: EDUTELLA: A P2P Networking Infrastructure based on RDF. In: Proceedings of the 11th World Wide World Conference (WWW 2002), Honolulu, Hawaii, USA, pp. 604–615 (2002)
56. Nejdl, W., Wolf, B., Staab, S., Tane, J.: Semantic Web Workshop 2002. CEUR Workshop Proceedings, vol. 55 (2002)
57. Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M., Brunkhorst, I., Loser, A.: Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-To-Peer Networks. In: Proceedings of the 12th WWW Conference, Budapest, Hungary (May 2003)
58. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. 19(1), 91–113 (2010)
59. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 3rd edn. Springer (2011)
60. Paoli, J., Yergeau, F., Sperberg-McQueen, M., Bray, T., Maler, E.: Extensible markup language (XML) 1.0. W3C recommendation, W3C, 5th edn. (November 2008), `http://www.w3.org/TR/2008/REC-xml-20081126/`
61. Papailiou, N., Konstantinou, I., Tsoumakos, D., Karras, P., Koziris, N.: H2RDF+: High-performance distributed joins over large-scale RDF graphs. In: BigData Conference (2013)

62. Patel-Schneider, P., Hayes, P.: RDF 1.1 semantics. W3C recommendation, W3C (February 2014), `http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/`
63. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. ACM Transactions on Database Systems 34(3), 16:1–16:45 (2009)
64. Punnoose, R., Crainiceanu, A., Rapp, D.: Rya: A Scalable RDF Triple Store for the Clouds. In: Workshop on Cloud Intelligence (in Conjunction with VLDB) (2012)
65. Rakhmawati, N.A., Umbrich, J., Karnstedt, M., Hasnain, A., Hausenblas, M.: Querying over Federated SPARQL Endpoints - A State of the Art Survey. CoRR, abs/1306.1723 (2013)
66. Raman, V., Attaluri, G.K., Barber, R., Chainani, N., Kalmuk, D., KulandaiSamy, V., Leenstra, J., Lightstone, S., Liu, S., Lohman, G.M., Malkemus, T., Müller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A.J., Zhang, L.: Db2 with blu acceleration: So much more than just a column store. PVLDB 6(11), 1080–1091 (2013)
67. Ravindra, P., Kim, H., Anyanwu, K.: An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part II. LNCS, vol. 6644, pp. 46–61. Springer, Heidelberg (2011)
68. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling Churn in a DHT. In: USENIX Annual Technical Conference (2004)
69. Rohloff, K., Schantz, R.E.: Clause-Iteration with MapReduce to Scalably Query Datagraphs in the SHARD Graph-Store. In: Workshop on Data-intensive Distributed Computing (2011)
70. Rowstron, A., Druschel, P.: Pastry: Scalable, Distributed Object Location and Routing for Large-Scale- Peer-to-Peer Storage Utility. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
71. Sakr, S., Liu, A., Fayoumi, A.G.: The Family of Mapreduce and Large-scale Data Processing Systems. ACM Comput. Surv. 46(1), 11:1–11:44 (2013)
72. Saleem, M., Khan, Y., Ivan Ermilov, A.H.A.D., Ngomo, A.-C.N.:
73. Schätzle, A., Przyjaciel-Zablocki, M., Lausen, G.: PigSPARQL: Mapping SPARQL to Pig Latin. In: SWIM (2011)
74. Schlosser, M.T., Sintek, M., Decker, S., Nejdl, W.: HyperCuP - Hypercubes, Ontologies and Efficient Search on Peer-to-peer Networks. In: Moro, G., Koubarakis, M. (eds.) AP2PC 2002. LNCS (LNAI), vol. 2530, pp. 112–124. Springer, Heidelberg (2003)
75. Schwarte, A., Haase, P., Hose, K., Schenkel, R., Schmidt, M.: Fedx: Optimization techniques for federated query processing on linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 601–616. Springer, Heidelberg (2011)
76. SHA-1. Secure hash standard. National Institute of Standards and Technology. Publication 180-1 (1995)
77. Shao, B., Wang, H., Li, Y.: The Trinity Graph Engine. Technical report (2012), `http://research.microsoft.com/pubs/161291/trinity.pdf`
78. Sidirourgos, L., Kokkinidis, G., Dalamagas, T., Christophides, V., Sellis, T.: Indexing Views to Route Queries in a PDMS. Journal of Distributed Parallel Databases 23, 45–68 (2008)
79. Staab, S., Stuckenschmidt, H. (eds.): Semantic Web and Peer-to-Peer: Decentralized Management and Exchange of Knowledge and Information. Springer (2006)
80. Stein, R., Zacharias, V.: RDF On Cloud Number Nine. In: Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic (May 2010)

81. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. IEEE/ACM Transactions on Networking 11(1), 17–32 (2003)
82. Triantafillou, P., Xiruhaki, C., Koubarakis, M., Ntarmos, N.: Towards high-performance peer-to-peer content and resource sharing systems. In: Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003) (January 2003)
83. Wilkinson, K.: Jena property table implementation. In: SSWS (2006)
84. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A Distributed Graph Engine for Web Scale RDF Data. In: PVLDB (2013)
85. Zhang, X., Chen, L., Tong, Y., Wang, M.: EAGRE: Towards Scalable I/O Efficient SPARQL Query Evaluation on the Cloud. In: ICDE (2013)
86. Zhang, X., Chen, L., Wang, M.: Towards Efficient Join Processing over Large RDF Graph Using MapReduce. In: Ailamaki, A., Bowers, S. (eds.) SSDBM 2012. LNCS, vol. 7338, pp. 250–259. Springer, Heidelberg (2012)