Chapter 1
Storing and Indexing Massive RDF Data Sets

Yongming Luo, François Picalausa, George H.L. Fletcher, Jan Hidders, and Stijn Vansummeren

Abstract In this chapter we present a general survey of the current state of the art in RDF storage and indexing. In the flurry of research on RDF data management in the last decade, we can identify three different perspectives on RDF: (1) a relational perspective; (2) an entity perspective; and (3) a graph-based perspective. Each of these three perspectives has drawn from ideas and results in three distinct research communities to propose solutions for managing RDF data: relational databases (for the relational perspective); information retrieval (for the entity perspective); and graph theory and graph databases (for the graph-based perspective). Our goal in this chapter is to give an up-to-date overview of representative solutions within each perspective.

1.1 Introduction: Different Perspectives on RDF storage

The Resource Description Framework (RDF for short) provides a flexible method for modeling information on the Web [34, 40]. All data items in RDF are uniformly represented as triples of the form *(subject, predicate, object)*, sometimes also referred to as *(subject, property, value)* triples. As a running

Yongming Luo
Eindhoven University of Technology, The Netherlands, e-mail: y.luo@tue.nl

François Picalausa
Université Libre de Bruxelles, Belgium, e-mail: fpicalau@ulb.ac.be

George H.L. Fletcher
Eindhoven University of Technology, The Netherlands, e-mail: g.h.l.fletcher@tue.nl

Jan Hidders
Delft University of Technology, The Netherlands, e-mail: a.j.h.hidders@tudelft.nl

Stijn Vansummeren
Université Libre de Bruxelles, Belgium, e-mail: stijn.vansummeren@ulb.ac.be

{⟨work5678, FileType, MP3 ⟩,
⟨work5678, Composer, Schoenberg ⟩,
⟨work1234, MediaType, LP ⟩,
⟨work1234, Composer, Debussy ⟩,
⟨work1234, Title, La Mer ⟩,
⟨user8604, likes, work5678 ⟩,
⟨user8604, likes, work1234 ⟩,
⟨user3789, name, Umi ⟩,
⟨user3789, birthdate, 1980 ⟩,
⟨user3789, likes, work1234⟩,
⟨user8604, name, Teppei ⟩,
⟨user8604, birthdate, 1975 ⟩,
⟨user8604, phone, 2223334444 ⟩,
⟨user8604, phone, 5556667777 ⟩,
⟨user8604, friendOf, user3789 ⟩,
⟨Debussy, style, impressionist⟩,
⟨Schoenberg, style, expressionist⟩, . . . }

Fig. 1.1 A small fragment of an RDF data set concerning music fans.

example for this chapter, a small fragment of an RDF dataset concerning music and music fans is given in Figure 1.1.

Spurred by efforts like the Linking Open Data project [35, 75], increasingly large volumes of data are being published in RDF. Notable contributors in this respect include areas as diverse as the government [11], the life sciences [9], web 2.0 communities, and so on.

To give an idea of the volumes of RDF data concerned, as of September 2010 there are 28,562,478,988 triples in total published by data sources participating in the Linking Open Data Project [5]. Many individual data sources (like, e.g., PubMed, DBPedia, MusicBrainz) contain hundreds of millions of triples (797,000,000; 672,000,000; and 178,775,789; respectively). These large volumes of RDF data motivate the need for scalable native RDF data management solutions capabable of efficiently storing, indexing, and querying RDF data.

In this chapter we present a general and up-to-date survey of the current state of the art in RDF storage and indexing.

It is important to note that RDF data management has been studied in a variety of contexts. This variety is actually reflected in a richness of the perspectives and approaches to storage and indexing of RDF datasets, typically driven by particular classes of query patterns and inspired by techniques developed in various research communities. In the literature, we can identify three basic perspectives underlying this variety.

The relational perspective. The first basic perspective, put forward by the database community, is that an RDF graph is just a particular type of relational data, and that techniques developed for storing, indexing and answering queries on relational data can hence be reused and specialized for

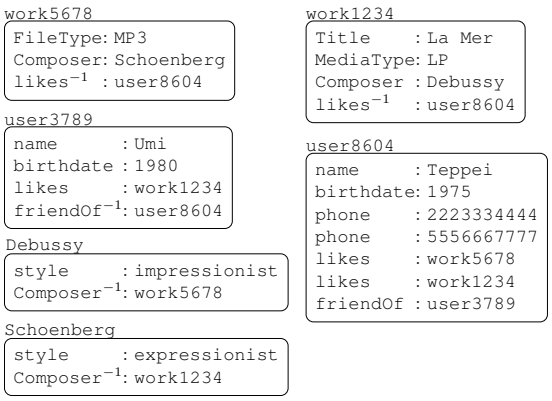

Fig. 1.2 The entity view of the music dataset in Figure 1.1

storing and indexing RDF graphs. In particular, techniques developed under this perspective typically aim to support the full SPARQL language. The most naive approach in this respect is simply to store all RDF triples in a single table over the relation schema *(subject, predicate, object)*. An important issue in this approach is that, due to the large size of the RDF graphs and the potentially large number of self-joins required to answer queries, care must be taken to devise an efficient physical layout with suitable indexes to support query answering. In addition to this simple *vertical representation*, there also exists a *horizontal representation* where the triple predicate values are interpreted as column names in a collection of relation schemas. For example, we can divide the music fan dataset of Figure 1.1 into five relations: `Works`, `Users`, `Composers`, `Likes`, and `FriendOf`; the `Users` relation would have columns `Name`, `BirthDate`, and `Phone`. Major issues here are dealing with missing values (e.g., subject `work5678` does not have a title) and multi-valued attributes (e.g., `user8604` has two phone numbers). In Section 1.2, we give an up to date survey of the literature on the relational perspective, thereby complementing earlier surveys focusing on this perspective [36, 50, 62, 64–66, 69].

The entity perspective. The second basic perspective, originating from the information retrieval community, is resource-centric. Under this *entity perspective*, resources in the RDF graph are interpreted as "objects", or "entities". Similarly to the way in which text documents are determined by a set of keyword terms in the classical information retrieval setting, each entity is determined by a set of attribute-value pairs in the entity perspective [18, 78]. In particular, a resource r in RDF graph G is viewed as an entity with the following set of (attribute, value) pairs:

$$entity(r) = \{(p, o) \mid (r, p, o) \in G\} \cup \{(p^{-1}, o) \mid (o, p, r) \in G\}.$$

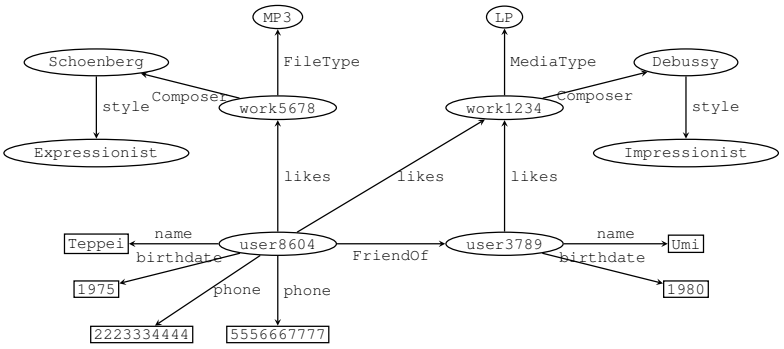

Fig. 1.3 The graph view of the music dataset in Figure 1.1

Figure 1.2 illustrates this idea for the RDF graph in Figure 1.1. Techniques from the information retrieval literature can then be specialized to support queries patterns that retrieve entities based on particular attributes and/or values [18]. For example, we have in Figure 1.2 that `user8604` is retrieved when searching for entities born in 1975 (i.e., have `1975` as a value on attribute `birthdate`) as well as when searching for entities with friends who like Impressionist music. Note that entity `user3789` is not retrieved by either of these queries. Further discussion and a survey of the literature on this perspective are given in Section 1.3.

The graph-based perspective. The third basic perspective, originating from research in semi-structured and graph databases, relaxes the entity-centric interpretation of resources in the entity view, focusing instead on the structure of the RDF data. Under this *graph-based perspective*, the focus is on supporting navigation in the RDF graph when viewed as a classical graph in which subjects and objects form the nodes, and triples specify directed, labeled edges. Under this perspective, the RDF graph of Figure 1.1 is hence viewed as the classical graph depicted in Figure 1.3. Typical query patterns supported in this perspective are graph-theoretic queries such as reachability between nodes. For example, in the graph of Figure 1.3, we might be interested to find those users which have a fan of `Debussy` somewhere in their social network (i.e., in the `friendOf` subgraph). The major issue under this perspective is how to explicitly and efficiently store and index the implicit graph structure. Further details and a survey of the literature on the graph-based perspective are given in Section 1.4.

Proviso. We note that recent research has devoted considerable effort to the study of managing massive RDF graphs in distributed environments such as P2P networks [58] and the MapReduce framework [63]. In this chapter, in contrast, we focus on storage and indexing techniques for a single RDF graph instance in a non-distributed setting. In addition, we focus on the storage and indexing of *extensional* RDF data only, and refer to other surveys [43, 50, 62,

69] and Chapter 11 for an overview of the state of the art in inferring and reasoning with the *intentional* data specified by RDF ontologies.

We assume that the reader is familiar with the basic concepts and terminology of the relational data model; as well as basic concepts of the RDF data model [34, 40]; and the SPARQL query language for RDF [59].

There are obviously tight connections between each of the above perspectives since they are interpretations of the same underlying data model. It is therefore sometimes difficult to clearly identify to which perspective a particular storage or indexing approach has the most affinity. Where appropriate, we indicate such "hybrids" in our detailed survey of the literature, to which we turn next.

1.2 Storing and Indexing under the Relational Perspective

Conceptually, we can discern two different approaches for storing RDF data in relational databases. The *vertical representation* approach stores all triples in an RDF graph as a single table over the relation schema *(subject, predicate, object)*. The *horizontal representation* approach, in contrast, interprets triple predicate values as column names, and stores RDF graphs in one or more wide tables. Additional indexes can be built on top of these representations to improve performance. We survey both representations next, and conclude with indexing.

1.2.1 A note on the storage of IRIs and literal values

Before we dive into the details of the vertical and horizontal representations, it is important to note that both approaches generally make special provisions for storing RDF resources efficiently. Indeed, rather than storing each IRI or literal value directly as a string, implementations usually associate a unique numerical identifier to each resource and store this identifier instead. There are two motivations for this strategy. First, since there is no a priori bound on the length of the IRIs or literal values that can occur in RDF graphs, it is necessary to support variable-length records when storing resources directly as strings. By storing the numerical identifiers instead, fixed-length records can be used. Second, and more importantly, RDF graphs typically contain very long IRI strings and literal values that, in addition, are frequently repeated in the same RDF graph. The latter is illustrated in Figure 1.1, for example, where `user8604` is repeated 8 times. Since this resource would be represented by a IRI like `http://www.example.org/users/user8604` in a

real-world RDF graph, storing the short numerical identifiers hence results in large space savings.

Unique identifiers can be computed in two general ways:

- Hash-based approaches obtain a unique identifier by applying a hash function to the resource string [30, 31, 64], where the hash function used for IRIs may differ from the hash function used for literal values. This is the approach taken by 3-Store [31], for example, where the hash functions are chosen in such a way that it is possible to distinguish between hash values that originate from IRIs and literal values. Of course, care must be taken to deal with possible hash collisions. In the extreme, the system may reject addition of new RDF triples when a collision is detected [31].
  To translate hash values back into the corresponding IRI or literal value when answering queries, a distinguished *dictionary* table is constructed.
- Counter-based approaches obtain a unique identifier by simply maintaining a counter that is incremented whenever a new resource is added. To answer queries, dictionary tables that map from identifiers to resources and vice versa are constructed [7, 32, 33, 56, 57]. Typically, these dictionary tables are stored as BTrees for efficient retrieval.
  A variant on this technique that is applicable when the RDF graph is static, is to first sort the resource strings in lexicographic order, and to assign the identifier n to the nth resource in this order. In such a case, a single dictionary table suffices to perform the mapping from identifiers to resources and vice versa [77].

Various optimisations can be devised to further improve storage space. For example, when literal values are small enough to serve directly as unique identifiers (e.g., literal integer values), there is no need to assign unique identifiers, provided that the storage medium can distinguish between the system-generated identifiers and the small literal values [15, 16]. Also, it is frequent that many IRIs in an RDF graph share the same namespace prefix. By separately encoding this namespace prefix, one can further reduce the storage requirements [15, 16, 49].

1.2.2 Vertical Representation

In the vertical representation (sometimes also called *triple store*), triples are conceptually stored in a single table over the relation schema *(subject, predicate, object)* [7, 10, 30, 31, 80]. Some implementations include an additional *context* column in order to store RDF *datasets* rather than single RDF graphs. In this case, the *context* column specifies the IRI of the named graph in which the RDF triple occurs.

An important issue in the vertical representation is that, due to the large size of the RDF graphs and the potentially large number of self-joins required

to answer queries, care must be taken to devise an efficient physical layout with suitable indexes to support query answering [15, 32, 33, 41, 49, 56, 57, 67, 79].

Unclustered indexes One of the first proposals for addressing this scalability issue compared the impact of adding the following four different sets of BTree indexes on the triple table [49]:

1. The first set of unclustered indexes consists of an index on the subject column (s) alone; an index on the property (p) column alone, and index on the object column (o) alone.
2. The second set of unclustered indexes consists of a combined index on subject and property (sp), as well as an index on the object column (o) alone.
3. The third set of unclustered indexes consists of a combined index on property and object (po).
4. The final set has a combined clustered index on all columns together (spo).

The authors note that, for an ad-hoc set of queries and on a specific RDF graph, the first set of indexes drastically improves the SQL-based query processing performance, whereas the other index sets achieve less performance.

More recently, Kolas et al. extend the idea of indexing each of the subject, predicate, and object columns separately [41]. In their Parliament database system, each RDF resource r (IRI or literal) is mapped, by means of a BTree, to a triple of pointers (p_1, p_2, p_3). Here, p_1 points to the first record in the triple table in which r occurs as a subject; p_2 points to the first record in which r occurs as a predicate; and similarly p_3 points to the first record in which r occurs as an object. In addition, each record in the triple table is extended with three additional pointers that point to the next triple in the table containing the same subject, predicate, and object, respectively. In this way a linked list is constructed. While this layout consumes very little storage space, traversing large linked lists may incur a non-negligible disk I/O cost.

Clustered BTree indexes Whereas the above indexes are *unclustered* and hence simply store pointers into the triple table (avoiding storing multiple copies of the RDF triples), there are many triple stores that aggressively store the vertical representation table in multiple sorted orders according to various permutations of the sequence *(context, subject, predicate, object)* and its subsequences [10, 15, 56, 57, 79]. Usually, a *clustered* BTree is constructed for each of the selected permutations for efficient retrieval; the desired triple ordering is then available in the leaves of the BTree. While these approaches are more demanding in terms of storage space, they support efficient query answering since the availability of the various sorted versions of the triple table enables fast merge joins.

- Weiss et al [79] consider this approach in their HexaStore engine by creating 6 different clustered BTree indexes: spo, sop, pso, pos, osp, and ops.

Additionally, common payload spaces are shared between indexes. For example, the *spo* and *pso* indexes share a common materialized set of associated *o* values. In this way, some storage redundancy is eliminated. Due to the remaining redundancy in the indexes, efficient retrieval of both fully and partially specified triples is supported. For example, it suffices to consult the *sop* index to retrieve all triples with subject `User8604` and object `user3789`. Recently, Wang et al. proposed to implement these indexes by means of a column-oriented database instead of using BTrees [77].

- Neumann and Weikum [56, 57] take this approach further in their RDF-3X engine by adding, to the 6 indexes above, so-called *projection indexes* for each strict subset of $\{subject, predicate, object\}$, again in every order. This adds an additional 9 indexes: *s*, *p*, *o*, *sp*, *ps*, *so*, *os*, *op*, and *po*. Instead of storing triples, the projection indexes conceptually map search keys to the number of triples that satisfy the search key. For example, the projection index on subject alone maps each subject *s* to the cardinality of the multiset

$$\{| (p, o) \mid (s, p, o) \text{ occurs in the RDF Graph } |\}.$$

  The projection indexes are used to answer aggregate queries; to avoid computing some intermediate join results; and to provide statistics that can be used by the RDF-3X cost-based query optimizer. RDF-3X also includes advanced algorithms for estimating join cardinalities to facilitate query processing.
  To reduce the storage space required, RDF-3X in addition uses a compression scheme at the BTree leaf block level. The idea behind this compression scheme is simple: in a given index sort order, say *spo*, it is likely that subsequent triples share a common subject or subject-predicate prefix. In this case, these repetitions are not stored.
- The widely used Virtuoso DBMS and RDF engine also stores different permutations of *spoc* [15, 16] (where *c* indicates the context column). By default, however, only clustered BTree indexes on *cspo* and *ocps* are materialized, although other indexes can be enabled by the database administrator. The storage cost for these indexes is reduced by introducing compression schemes at the data level, as well as at the page level.
  In addition, Erling and Mikhailov [16] report that in practical RDF graphs, many triples share the same predicate and object. To take advantage of this property, Virtuoso builds bitmap indexes for each *opc* prefix by default, storing the various subjects. McGlothlin et al. build on this idea and use bitmap indexes instead of BTrees as the main storage structure [53].
  Furthermore, Atre et al. [4] build a virtual 3D bitmap upon an RDF graph, with each dimension representing the subjects, predicates and objects, respectively. Each cell of this 3D bitmap hence states the existence of one combination of the coordinates, i.e., a triple. By slicing the bitmap from different dimensions, several 2D bit-matrices (BitMats) are generated, es-

sentially indicating various permutations of the triples. These so-called *BitMats* are compressed and stored on disk, along with a meta-file maintaining summary and location information. Bit-operations on this meta-file are used to filter out BitMats during join evaluation, thereby accelerating query evaluation.

- Finally, Fletcher and Beck [17] propose a hybrid of the unclustered and clustered approaches, wherein a single BTree index is built over the set of all resources appearing in the dataset. The payload for a resource r consists of three sorted lists, namely *op* where r appears as a subject, *so* where r appears as a predicate, and *sp* where r appears as an object. The advantage of this so-called TripleT approach is simplicity of design and maintenance, as well as increased locality of data, in the sense that all information pertaining to a resource is available in a single local data structure. The TripleT index was shown to be competitive with the previous approaches, in terms of both storage and query evaluation costs.

Variations on the Clustered index approach Predating the above approaches, Harth et al. [32, 33] considered clustered BTree indexing of *spoc* quads in their Yars system. Since they were not interested in answering queries involving joins, only 6 of the 16 subsets of *spoc* are materialized: *spoc*, *poc*, *ocs*, *csp*, *cp*, *os*. In more recent work, Harth et al. alternatively consider the use of extensible hash-tables and in-memory sparse indexes instead of BTrees [33]. They observe that while disk I/O decreases from $O(log(n))$ to $O(1)$ to answer queries on n quads by using extensible hash tables rather than BTrees, 16 hash-tables are required to mimic the behaviour of the 6 BTrees. As an alternative to hash tables, Harth et al. note that a similar decrease in disk I/O can be obtained by constructing an in-memory sparse index for each of the 6 subsets of *spoc*. Each entry in a sparse index points to the first record of a sorted block on disk. Evidently, there is a trade-off between main memory occupation and performance: to accommodate larger RDF graphs or to reduce main memory consumption, the index can be made sparser by making the sorted blocks larger, at the expense of more disk I/O.

Wood et al. [82] build on this idea by representing the six sparse indexes on *spoc* using AVL trees [82]. For performance reasons, nodes of the AVL tree point to separate pages of 256 ordered triples. Each node of the AVL tree stores the first and the last triples of the page it represents to enable traversal of the tree without accessing the pages.

1.2.3 Horizontal Representation

Under the horizontal representation, RDF data is conceptually stored in a single table of the following format: the table has one column for each predicate value that occurs in the RDF graph, and one row for each subject value. For each (s, p, o) triple, the object o is placed in the p column of row s. The

music fan data from Figure 1.1 would hence be represented by the following table.

subject	FileType	Composer	...	phone	friendOf	style
work5678	MP3	Schoenberg				
work1234		Debussy				
...						
user8604				{2223334444, 5556667777}	user3789	
Debussy						impressionist
Schoenberg						expressionist

As can be seen from this example, it is rare that a subject occurs with all possible predicate values, leading to sparse tables with many empty cells. Care must hence be taken in the physical layout of the table to avoid storing the empty cells. Also, since it is possible that a subject has multiple objects for the same predicate (e.g., `user8604` has multiple phone numbers), each cell of the table represents in principle a *set* of objects, which again must be taken into account in the physical layout.

Property tables To minimize the storage overhead caused by empty cells, the so-called *property-table approach* concentrates on dividing the wide table in multiple smaller tables containing related predicates [46,68,80]. For example, in the music fan RDF graph, a different table could be introduced for `Works`, `Fans`, and `Artists`. In this scenario, the `Works` table would have columns for `Composer`, `FileType`, `MediaType`, and `Title`, but would not contain the unrelated `phone` or `friendOf` columns. The following strategies have been proposed to identify related predicates.

- In the popular Jena RDF framework, the grouping of predicates is defined by applications [80]. In particular, the application programmer must specify which predicates are multi-valued. For each such multi-valued predicate p, a new table with schema $(\mathit{subject}, p)$ is created, thereby avoiding replicating other values. Jena also supports so-called *property-class* tables, where a new table is created for each value of the `rdf:type` predicate. Again, the actual layout of each property-class table is application-defined. The remaining predicates that are not in any defined group are stored independently.
- In RDFBroker, Sintek et al. [68] identify related predicates by computing, for every subject x in the RDF graph G the set of predicates $P_x = \{p \mid \exists o, ((x, p, o) \in G)\}$. For each such set P_x a corresponding predicate table over the relational schema $\{\mathit{subject}\} \cup P_x$ is created. Queries can be answered by first identifying the property tables that have all of the predicate values specified in the query; answering the query on these tables; and then taking the union of the results.
  Sintek et al. note that this approach actually creates many small tables, which is harmful for query evaluation performance. To counter this, they propose various criteria for merging small tables into larger ones, introducing NULL values when a predicate is absent.

- Levandoski et al. [46] leverage previous work on association rule mining to automatically determine the predicates that often occur together, through a methodology they call data-centric storage. The methodology aims at maximizing the size of each group of predicates, while minimizing the number of NULL values that will occur in the tables. The remaining predicates that are not in any group are stored independently.

Vertical partitioning The so-called *vertically partitioned database approach* (not to be confused with the vertical representation approach of Section 1.2.2) takes the decomposition of the horizontal representation to its extreme [1]: each predicate column p of the horizontal table is materialized as a binary table over the schema $(subject, p)$. Each row of each binary table essentially corresponds to a triple. Note that, hence, both the empty cell issue and the multiple object issue are solved at the same time. Abadi et al [1] and Sidirourgos [67] note that the performance of this approach is best when sorting the binary tables lexicographically according to $(subject, p)$ to allow fast joins; and that this approach is naturally implemented by a column oriented database.

Disadvantages and advantages It has been observed that, in both approaches, the distinction between predicate values (which are elevated to the status of column names) and subject and object values (which are stored as normal values) is a weakness when answering queries that do not specify the predicate value. For such queries, the whole horizontal table (or all of its partitions) must be analyzed. This hinders performance, especially in the presence of many predicates values. Moreover, the relational schema must be changed whenever a new predicate value is added to the RDF graph. This is not well supported in relational database management systems, where relation schemas are traditionally considered to be static.

On the positive side, the horizontal representation makes it easy to support typing of object values (e.g., it becomes possible to declare that object values of predicate `age` are integers, whereas object values of predicate `birthdate` are dates). Moreover, it is easy to integrate existing relational data with RDF data: it suffices to consider the key of a relational table as the subject, and the remaining columns as predicates. As such, the existing table essentially becomes a new RDF graph [80].

1.2.4 More on Indexing

While some of the storage strategies presented in Sections 1.2.2 and 1.2.3 already provide storage strategy-dependent indexes, additional indexes may be desired to speed up query processing even further.

Literal-specific indexes A first line of work in this respect targets the indexation of the *literals* occurring in an RDF graph. To support full-text

search, for example, one can add inverted indexes on string literals [15, 32], or N-grams indexes [45] to support regular expression search. Similarly, if it is known that particular literals hold geographical information (e.g., latitude and longitude), specialized spatial indexes such as kd- and quad-trees can be added [42].

Join indexes A second line of work extends the classical idea of join indexes [74] to the RDF setting. In particular, Chong et al. [10] propose the construction of 6 binary join indexes to speed up self-joins in the vertical representation storage approach: there is an index for the self-join on subject-subject, on subject-predicate, on subject-object, on predicate-predicate, on predicate-object, and on object-object. In this way, a SPARQL query that expresses a join between two triple patterns, like

```
SELECT ?work
WHERE {
  ?user <likes>    ?work     .
  ?work <Composer> "Debussy" .
}
```

can be answered using the object-subject join index.

Groppe et al. [25] extend this idea further. They remark in particular that triple patterns in SPARQL queries rarely consist of variables only, but also mention IRIs or literals (as in the example above). In order to find the joining triples that satisfy the mentioned IRI/literal requirements, Groppe et al. extend the notion of a join index and store, for each of the possible self-joins listed above (subject-subject, subject-predicate, ...) 16 different indexes. Conceptually, each of the latter indexes map sequences of IRIs and literals that do not occur in the join position to the pair of joining triples.

To illustrate, consider the join index on subject-object which is meant to aid in answering joins of two triple patterns, say $(?x, p_1, o_1)$ and $(s_2, p_2, ?x)$ like in the example query above. Groppe et al. construct the 16 indexes that map all subsets of p_1, o_1, s_2, p_2 to the joining triples. So, in an RDF graph consisting of the triples

tid	subject	predicate	object
t_1	a	b	c
t_2	d	e	a
t_3	f	g	d

the 16 indexes would be constructed as follows.

p_1	o_1	s_2	p_2	
b	c	d	b	(t_1, t_2)
b	a	f	g	(t_2, t_3)

...

p_1	s_2	p_2	
b	d	b	(t_1, t_2)
b	f	g	(t_2, t_3)

...

p_1	
b	(t_1, t_2)
b	(t_2, t_3)

This approach can be further extended to handle joins between more than two triple patterns.

Materialized views Finally, a third line of work applies the classical notion of answering queries using views [28] to the RDF setting, thereby also extending the work on join indexes discussed above.

In a nutshell, when answering queries using views we are given a query Q over an RDF graph G, as well as a set of materialized (relational) views $V_1, \ldots, V_n$ over G and we want to find the cheapest query execution plan for Q that can use both G and the views $V_1, \ldots, V_n$. Of course, a critical difficulty in this respect is the selection of the views $V_1, \ldots, V_n$ to materialize.

In the `RDF_MATCH` approach, Chong et al. [10] propose to materialize so-called *Subject-Property tables* in addition to the (vertically represented) RDF graph. A Subject-Property table is similar to the property-tables from the horizontal representation storage approach: for a set of subject values and a group of related single-valued predicates, a table is constructed where each predicate occurs as a column name. Each row contains one subject value, along with the object value of each predicate. Contrary to property-tables of the horizontal store approach, however, the proposed view here is not the primary way of storage.

In the RDFMatView system, Castillo et al. [8] offer the users the possibility to identify and materialize SPARQL basic graph pattern. They note that basic graph patterns can be viewed as conjunctive queries, for which query containment is known to be decidable. When processing a new query, materialized views that can be used to answer the query are selected based on a query containment test. Subsequently, the selected views are assembled to help in the processing. A corresponding cost model to compare alternate rewritings is also proposed. A similar approach uses expression trees of the input expression as the basis for caching intermediate queries results [85].

The authors of RDFViewS [22] take the complementary approach of identifying important conjunctive queries in a given weighted workload. An RDF Schema is used to guide the process of selecting the appropriate conjunctive queries.

1.3 Storing and Indexing under the Entity Perspective

As mentioned earlier in Section 1.1, a major alternative way to view an RDF graph is to treat it as a collection of entity descriptions [18, 78]. Similarly to the way in which text documents are viewed as sets of keyword terms in the classical information retrieval setting, each entity is determined by a set of attribute-value and relationship-entity pairs in the entity perspective. Approaches under the entity perspective make heavy use of the inverted index data structure [81]. The reason for this is two-fold. First and foremost, inverted indexes have proven to scale to very large real-world systems. Moreover, many of the aspects of inverted indexes have been thoroughly studied

(e.g., encoding and compression techniques), from which the new approaches for RDF storage and indexing can benefit.

A large part of the work under the entity perspective investigates how to map the RDF model to traditional information retrieval (IR) systems. Typically, the following two general types of queries are to be supported:

- *Simple keyword queries.* A keyword query returns all entities that contain an attribute, relationship, and/or value relevant to a given keyword. To illustrate, in Figure 1.2, if the keyword query is "work5678", query evaluation returns the set of all entities having `work5678` somewhere in their description: namely, the entities `work5678`, `Debussy` and `user8604`.
- *Conditional entity-centric queries.* A conditional entity-centric query returns all known entities that satisfy some given conditions on a combination of attribute, relationships, and values at the same time. For Figure 1.2, the query "retrieve all entities with the value `work5678` on the $\texttt{Composer}^{-1}$ attribute" is such a conditional entity-centric query. This query returns the set of entities having value `work5678` on the $\texttt{Composer}^{-1}$ attribute, namely, the entity `Debussy`.

Note that, in contrast to relational approaches, query results under the entity perspective return a subset of the known entities, rather than relational tables. In this respect, the relational queries are more powerful since they allow the restructuring and combination of data (e.g., by performing joins etc.), whereas the entity-centric queries only return data in their existing form. It should be noted, however, that set operations (intersection, union, etc.) are usually supported in the entity perspective to further manipulate the returned set of entities [18]. A nice comparison of index maintenance costs between relational and entity perspective systems is presented in [12].

In the rest of this section we will introduce some of the representative works in the literature. Note that this is not an exhaustive survey of this rapidly advancing research area, but rather a selection of recent systems that illustrate the basic concepts of storage and indexing RDF data under the entity perspective.

1.3.1 Dataspaces

Dong and Halevy [13] were the first to consider inverted indexes for answering entity-centric queries over RDF graphs, in the context of the so-called *dataspace support system paradigm*. First proposed by Franklin et al. [20,29], the dataspaces paradigm constitute a gradual, pay-as-you go, approach to the integration of information from across diverse, interrelated, but heterogeneous data sources. In this setting, dataspaces should also provide query support in a pay-as-you go setting, depending on the querying capabilities of the data sources themselves. For example, if a data source is plain text,

maybe only keyword search is enabled. If it is semi-structured data, a corresponding structured query language (comparable to XQUERY or SPARQL) could be supported. This also requires the underlying index structure to be flexible enough to accommodate the query language.

While Dong and Halevy [13] do not explicitly use the terms *RDF data model* and *RDF graphs*, the data model that they use is equivalent to RDF. Specifically, in their paper, data from different sources is unified in triples of the form $(entity, attribute, value)$ or $(entity, association, entity)$. These triples are then passed to the indexer for further processing. The proposed system supports both simple keyword queries and conditional entity-centric queries, and always returns a collection of entities. The system also provides support for synonyms and hierarchies, which supports the discovery of more relevant results than plain querying. Referring to Figure 1.2, for example, if the (static) system-defined hierarchy indicates that `MediaType` is a superclass of `FileType`, then the evaluation of the conditional entity-centric query "retrieve all entities with value `MP3` on the `MediaType` attribute" with hierarchy expansion contains the entity `work1234`.

The basic indexing structure of Dong and Halevy is an inverted index. Each *term* in this index points to an inverted list that contains all the identifiers of all entities that *match* the term. There are three kinds of terms:

- To start with every value v is considered a term. An entity e matches the term v if there exists a triple ($entity\ e, attribute\ a, value\ v$), for some attribute a. This is the same as the traditional IR approaches, to provide answering keyword query capability.
- To support conditional entity-centric queries, concatenations of a value v and attribute a (denoted $v//a//$) are also considered to be terms. An entity matches such a term if the triple ($entity\ e, attribute\ a, value\ v$) is present.
- Moreover, concatenations of value v and association a_1 ($v//a_1//$) are also considered terms. An entity e_1 matches this term if there exists a pair of triples ($entity\ e_1, association\ a_1, entity\ e_2$) and ($entity\ e_2, attribute\ a_2$, $value\ v$) for some entity id e_2 and some attribute a_2. These kinds of terms are designed to boost a specialized conditional entity-centric query, called the *one-step keyword query*. Such queries are meant to return all entities in which a given keyword appears, as well as entities that are related to entities in which the keyword appears.

For example, when indexing the example dataset in Figure 1.2, the values `(MP3)`, `(Schoenberg)` and `(user8604)` will be treated as terms (among others). The concatenations of values and attributes `(MP3//FileType//)` and `(Schoenberg//Composer//)` will also be considered as terms (among others). Then entity `work5678` matches both of these terms, and will hence appear in the inverted list of both terms. Furthermore, because of the triple `(user8604, likes, work5678)`, two associations `(user8604, likes, work5678)` and `(work5678, likes`$^{-1}$`, user8604)` will be

generated. For the first association, we will treat `(MP3//likes//)` and `(Schoenberg//likes//)` as terms pointing to `user8604`. For the latter association, we let the terms `(Teppei//likes`$^{-1}$`)`, `(1975//likes`$^{-1}$`)`, `(2223334444//likes`$^{-1}$`)`, and `(5556667777//likes`$^{-1}$`)` have the entity `work5678` in their inverted lists.

Hierarchies are supported using three methods: duplication, hierarchy path and a hybrid of these two. The idea is to materialize some of the queries as terms beforehand, so that query answering will be accelerated. The system has a built-in synonym table. By using this table, synonymous attributes are transformed to a canonical one in the index, and thereby related results are naturally returned.

Updates are quite expensive on this index, because one entity is distributed over several inverted lists. The authors suggest to group updates together and do batch update operations to ease I/O cost. An open issue from this approach is that the result highly relies on how well the information is extracted and matched with each other from the data sources, and how well the hierarchy and synonym information are maintained. These are also open problems in information retrieval research in general.

1.3.2 SIREn

SIREn is part of the Sindice search engine project from DERI, where the objective is to index the entire "Web of Data" as a Dataspace [12]. In SIREn, data is unified to a set of quadruples of the form of (*dataset*, *entity*, *attribute*, *value*), upon which a tree model is built. In this tree, the dataset is the root, then the entity, attribute, and value are the children of the previous element, respectively. Each node in the tree is encoded with some encoding scheme (Dewey Order for instance [24]), so that the relationship between two nodes in a tree can be determined by only looking at the encoding value. SIREn has a clear definition of logical operators on this tree model. All query answering is then done by translating these logical operators into physical operations on a representation of this tree.

SIREn extends traditional inverted indexes for physically representing these trees. This is done mainly by extending the inverted list component of the index. For indexing in SIREn, the search term could be the identifier of the dataset, the entity, the attribute or the value. The corresponding inverted list consists of five different streams of integers: a list of entity identifiers, of term frequencies, of attribute identifiers, of value identifiers and of term positions. Further refinement is done for each term to reduce the index size. Notice that each step of query answering does not return a collection of entities, but rather a collection of nodes in the tree model. Intermediate results can then be used as an input for subsequent operations.

To answer a keyword query, the system performs a lookup of the search term, then returns related terms with their inverted lists. The keyword query could come with a specified type, and then only the given type of inverted lists are returned. Partial matching is supported by the keyword search, such as prefix matching and regular expression, depending on the organization of the terms. To answer conditional entity-centric queries, the system first performs a keyword search with a specific type, then a join of two inverted lists is performed. Results can be further refined by other logical operations, such as projections.

Compression techniques are also incorporated in SIREn. Five inverted files are created complying with the five types of inverted lists discussed above. Each inverted file is constructed in a block-based fashion, and then various compression techniques are studied on these files. One limitation of SIREn is that join operations between data from different data sources are not efficiently supported.

1.3.3 Semplore

Semplore [76] is another representative approach of the entity perspective approach, also using inverted indexes as the storage back-end. Semplore works on RDF datasets, and supports hybrid queries combing of a subset of SPARQL and full text search.

Semplore has three indexes: an ontology index, an entity index and a textual index. The ontology index stores the ontology graph of the dataset, including super/sub concepts and super/sub relations. The entity index stores the relationships between the entities. The textual index handles all triples with the "text" predicate, enabling the system to do keyword search. The highlight of Semplore is to type RDF resources as Document, Field, and/or Term concepts in the inverted indexes. Moreover, the system uses a position list in inverted lists as another dimension of the inverted index. In particular, for each triple (s, p, o),

- For the ontology index, (p, o) is treated as a term and s is treated as a document. Here $p \in \{subConOf, superConOf, subRelOf, superRelOf, type\}$.
- For the entity index, two permutations (p, s, o) and (p^{-1}, o, s) are stored. Here p/p^{-1} is treated as a term and s/o is treated as a document. o/p is stored in the position list. For example in Figure 1.2, for entity `work5678` the following mapping is done:

Term	Document	Position List
`FileType`	`work5678`	`MP3`
`FileType`$^{-1}$	`MP3`	`work5678`
`Composer`	`work5678`	`Schoenberg`
`Composer`$^{-1}$	`Schoenberg`	`work5678`
`likes`	`user8604`	`work5678`
`likes`$^{-1}$	`work5678`	`user8604`

- For the textual index, (p, k) is treated as a term, where p is "text", and k is a token appearing in o. s is treated as the document. Following the previous example, if one more triple `(work5678, text, "wonderful description")` is added for entity `work5678`, then the words `wonderful` and `description` will both be treated as terms, having `work5678` in their inverted lists.

Semplore has three basic operators: basic-retrieval, merge-sort and mass-union. An input query is translated into these operations and performed on the index. The main idea is to traverse the query parse tree in a depth first way, combining intermediate results.

To be able to handle incremental updates, Semplore proposes a block based index structure, splitting inverted lists into blocks. The Landmark technique [48] is used to locate a document in one inverted list. When blocks are full, they can split as nodes split in a B-Tree. Single update and batch update are both supported in this case.

Semplore supports a richer query expressiveness than the Dataspace and SIREn approaches. It can answer queries concerning entity relations. However, this also involves more I/O cost, where the join operation becomes the dominant operation in the system. Also due to the incompleteness of the permutations, it is not possible to answer queries when predicates are missing (e.g. queries like (a,?,b)).

1.3.4 More on Information Retrieval techniques

Information Retrieval techniques are also widely used in other RDF systems. Here we mention some of them.

The most common use of IR techniques is using an inverted index to provide full text search functionality. In YARS2 [33], similar to the Dataspace approach, for each triple (s, p, o), the index uses the literals appearing in object position (o) as terms, pointing to a list of subjects as the inverted list. Some systems also benefit from wildcard/prefix search functionality from inverted indexes, e.g., [55, 82]. Natural language processing techniques may also be used for identifying keywords in the literal strings or URIs.

Another common use is object consolidation [72, 78], which is the merging of resources from different datasets as the same entity. Several techniques such as word cleaning and synonym merging are applied in this case.

Finally, we note that several systems (e.g., [14, 44, 71]) take a hybrid approach of supporting an entity perspective on the underlying RDF dataset, while translating search queries into structured queries on the data, typically organized under the graph-based perspective, which we discuss in the next section.

1.4 Storing and Indexing under the Graph-based Perspective

As the name *RDF graph* already hints at, the RDF data model can be seen as essentially a graph-based data model, albeit with special features such as nodes that can act as edge labels and no fundamental distinction between schemas and instances, which can be represented in one and the same graph. This graph-based nature implies that both the types of queries and updates that need to be supported as well as the types of data structures that can be used to optimize these, will be similar to those for graph databases [2]. Typical queries would for example be pattern matching, e.g., find an embedding of a certain graph, and path expressions, e.g., check if there is a certain type of path between two nodes. Graph databases, in turn, are similarly closely related to object-oriented and semi-structured databases. Indeed, many ideas and techniques developed earlier for those databases have already been adapted to the RDF setting. For example, Bönström et al. show in [6] that RDF can be straightforwardly and effectively stored in an object-oriented database by mapping both triples and resources to objects.

One of the key ideas that has been at the center of much research for graph-based and semi-structured data is the notion of *structural index*. This section is therefore organized as follows. We first discuss the application of general graph-based indexing techniques to RDF databases. This is followed by a section on structural indexes where we first present their historical development for graph-based and semi-structured databases; and then their application to RDF databases.

1.4.1 General Graph-based Indexing Methods

Suffix arrays The problem of matching simple path expressions in a labeled graph can be seen as a generalization of locating the occurrences of a string as a substring in a larger target string. A well-known indexing structure for the latter setting is that of the *suffix array*, which can be described as follows. Assuming that the lexicographically sorted list of all suffixes of the target string is $[s_1, \ldots, s_n]$ then a suffix array is an array $a[1..n]$ such that $a[i] = (s_i, j_i)$ with j_i the starting position of s_i in the target string. Given a search string s we can quickly find in the array all entries $1 \leq i \leq n$ where s is a prefix of s_i and therefore j_i is a position where s occurs. This idea can be generalized to node and edge labeled directed acyclic graphs as follows. The suffixes are here defined as alternating lists of node and edge labels that occur on paths that end in a leaf, i.e., a node with no outgoing edges. With each suffix the suffix array then associates instead of j_i a set of nodes in the graph. This approach is adapted and applied to RDF by Matono et al.

in [51] where classes are interpreted as node labels and predicates as edge labels. The authors justify their approach by claiming that in practical RDF graphs cycles are rare, but propose nonetheless an extension that can deal with cycles by marking in suffixes the labels that represent repeating nodes.

Tree labeling schemes A common technique for optimizing path queries in trees is to label the nodes in the trees in such a way that the topological relationship between the nodes can be determined by an easy comparison of the labels only. For example, for XML a common labeling scheme was introduced by Li and Moon [47]. They label each node with a tuple (*pre*, *post*, *level*), where *pre* is an integer denoting the position of the node in a pre-order tree walk, *post* is an integer denoting the position of the node in a post-order tree walk and *level* is the level of the node in the tree where the root is the lowest level. Two nodes with labels (pr_1, po_1, l_1) and (pr_2, po_2, l_2) will have an ancestor-descendant relationship iff $pr_1 < pr_2$ and $po_1 < po_2$. They have a parent-child relationship iff in addition $l_2 = l_1 + 1$. We refer to Gou and Chirkova [24] for an overview of such labeling schemes for XML. In principle, this technique can be also applied to directed acyclic graphs if we transform them into a tree by (1) adding a single new root node above the old root nodes and (2) duplicating nodes that have multiple incoming edges. Clearly this can lead to a very redundant representation, but it can still speed up certain path queries. This approach is investigated by Matono et al. in [52] where, like in [51], it is assumed that RDF graphs are mostly acyclic. Note that for some parts of the RDF graph such as the Class hierarchy and the Predicate hierarchy described by an RDF Schema ontology, this assumption is always correct. As such, these hierarchies are stored separately by Matono et al. [51, 52]. Further study of interval-based labeling schemes for RDF graphs with and without cycles has been undertaken by Furche et al. [21].

Distance-based indexing In [73], Udrea et al. propose a tree-shaped indexing structure called GRIN index where each node in the tree identifies a subgraph of the indexed RDF graph. This is done by associating with each node a pair (r, d) with a resource r and a distance d, defined as minimal path length and denoted here as $\delta_G(r, s)$ for the distance between r and s in graph G, which then refers to the subgraph that is spanned by the resources that are within distance d of r. The index tree is binary and built such that (1) the root is associated with all resources in the indexed graph, (2) the sets of indexed graph nodes associated with sibling nodes in the tree do not overlap and (3) the set of nodes associated with a parent is the union of those of its two children. As is shown in [73] such indexes can be built using fairly efficient clustering algorithms similar to partitioning around medoids.

Given a query like a basic graph pattern, we can try to identify the lowest candidate nodes in the tree that refer to subgraphs into which the pattern might match. There are two rules used for this, where we check for query q and an index node with pair (r, d) the following:

1. If q mentions resource s and $\delta_G(r, s) > d$, then the index node is not a candidate, and neither are all of its descendants.
2. If q contains also a variable v and $\delta_G(r, s) + \delta_q(s, v) > d$, then the index node is not a candidate and neither are any of its descendants. Note that this rule eliminates viable candidates because it is not necessarily true that if v is mapped to r_v by a matching of the query then $\delta_G(r, s) + \delta_q(s, v) = \delta_G(r, r_v)$. However, this rule was found to work well in practice in [73].

The query is then executed with standard main-memory-based pattern matching algorithms on the subgraphs associated with the lowest remaining candidates, i.e., those that have no parent in the index tree which is a candidate. The restriction to the lowest is correct if we assume that the basic graph pattern is connected and mentions at least one resource.

System *Π* Several graph-based indexing techniques were investigated by Wu et al. in the context of System *Π*, a hypergraph-based RDF store. In [83], which preceded system *Π*, Wu et al. propose to use an index based on Prüfer sequences. These sequences had already been suggested for indexing XML by Rao and Moon in [61] and can encode node-labeled trees into sequences such that the problem of finding embeddings of tree patterns is reduced to finding subsequences. However, since RDF graphs may contain cycles and have labeled edges, this method needs to be adapted considerably and loses much of its elegance and efficiency, which is perhaps why it was not used in System *Π* as presented by Wu et al. in [83]. A technique mentioned in both papers is the encoding of the partial orderings defined by the Class and Property hierarchies in RDF Schema ontologies using a labeling technique based on prime numbers. First, each node n in the partial order is assigned a unique prime number $p(n)$. Then, we define $c(n) = \Pi_{m \preceq n} p(m)$ where $\preceq$ denotes the partial order. It then holds that $n \preceq n'$ iff $c(n')$ is a multiple of $c(n)$.

1.4.2 Structural Indexes

An often recurring and important notion for indexing semi-structured data is the *structural index* (or *structure index* or *structural summary*, as it is also known). The simplest way to explain them is if we assume that the data instance is represented by a directed edge-labeled graph. An example of a structural index is then a reduced version of this graph where certain nodes have been merged while maintaining all edges. In the resulting graph we store with each node m the set $M(m)$ of nodes, also called the *payload* of m, that where merged into it. The crucial property of such an index is that for many graph queries, such as path expressions and graph patterns, it holds that they can be executed over the structural index and then give us a set of candidate results that contains the correct results as a subset. For example,

consider the query that returns all pairs of nodes (n_1, n_2) such that there is a directed path from n_1 to n_2 that satisfies a certain regular expression. If, when executed over a structural index, the pair of merged nodes (m_1, m_2) is returned then all pairs in $M(m_1) \times M(m_2)$ can be considered as candidate results. In fact, for certain types of queries and merging criteria it can be shown that the set of candidate results is always *exactly* the correct result, in which case the index is said to be *precise*. However, even if this is not the case, in which case the index is called *approximate*, the structural index can provide a quick way of obtaining a relatively small set of candidate solutions, especially if the original graph has to be stored on disk but the index fits in main memory.

Dataguides, 1-indexes, 2-indexes and *T*-indexes The first work that introduced structural indexes was that by Goldman and Widom [23] which introduced a special kind of structural index called the *dataguide*. In their setting, both the data instance and the structural index are rooted graphs. The dataguides have the restriction that each node has for each label at most one outgoing edge with that label. The focus is on queries defined by path expressions that are evaluated starting from the root. For certain path expressions and a particular type of dataguide called *strong dataguide* it is shown that the dataguide is precise. This work was extended by Milo and Suciu in [54] where three types of structural indexes where introduced: the *1-index*, the *2-index* and the *T-index*. The 1-index focuses on the same queries as dataguides, but the merging of nodes is based on an equivalence relation based on bisimulation that follows the edges in reverse, which can be relatively efficiently computed, but still produces a precise index. The 2-index focuses on queries of the form $* \; x_1 \; P \; x_2$ where $*$ is the wildcard path expression that matches any path and P is an arbitrary path expression. The variables x_1 and x_2 indicate the positions in the path that are returned as a result, so the result is a set of pairs of nodes. A T-index focuses on an even wider class of queries, viz., those of the form $T_1 \; x_1 \; T_2 \; x_2 \ldots T_n \; x_n$ where each T_i is either a path expression or a formula expressing a local condition. A different T-index is defined for each query template, where a query template is defined by choosing for certain T_i a particular path expression, and specifying for others only if they are a path expression or a formula. The 2-index and T-indexes are not based on merging on single nodes, as in dataguides and 1-indexes, but rather on tuples of n nodes or less, but also here there is an accompanying notion of equivalence and a result that says that these indexes are precise for the queries they are defined for.

Localized notions of (bi)similarity This work was applied and extended in an XML setting by Kaushik et al. in [39]. Here data instances are modeled as a node-labeled tree with additional `idref` edges that may introduce cycles. The authors observe that 1-indexes can become very large in practice, and that they can be reduced in size if we redefine the equivalence relation for merging by basing it on k-bisimulation which essentially considers only the

local environment of a node within distance k. This causes the equivalence relation to be coarser, therefore more nodes to be merged, and so produces a smaller index, called an A(k)-index. However, it also causes the resulting index to be only approximate and no longer precise for the considered query language (namely: regular path expressions). It is shown that this approximation is relatively rare in practice, and that it often can be detected that the result is precise without considering the original data instance graph.

This work was subsequently extended by Kaushik et al. in [37] for the subset of XPath known as BPQ, the *Branching Path Queries*. These queries allow the use of the forward axes `child` and `descendant`, the backward axes `parent` and `ancestor`, and navigating using the `idref` attributes both forwards and backwards. They also allow filtering conditions for intermediate nodes that consist of path expressions combined with the `and`, `or` and `not` operators. The basic trick is here to extend the instance graph with all the reverse edges, i.e., for a `child` edge we add a reverse `parent` edge, and for every `idref` edge we add a reverse $\texttt{idref}^{-1}$ edge. We then can again define a notion of equivalence based on forward and backward bisimulation, which now looks in both directions, and results in a precise index. Moreover, we can again reduce the size of the index by considering only the local environment of a node up to distance k and sacrificing precision.

Connections with query language fragments Since for structural indexes there is a trade-off between size and precision, i.e., a smaller index is probably less precise, and because efficiency requires that indexes are as small as possible, it is interesting to determine what the smallest index is that is still precise for a certain query language. It was for example shown by Kaushik et al. in [37] that for XML and the XPath subset BPQ the smallest precise structural index is defined based on forward and backward bisimulation. Similar results where obtained by Ramanan in [60] where it was shown that for the positive fragment of BPQ, i.e, without negation, and for TPQ (Tree Pattern Queries), not bisimulation but simulation defines the smallest precise structural index. Similar results for other fragments of XPath where obtained by Gyssens et al. in [26] and by Wu et al. in [84]. The usefulness of such analysis is shown by Fletcher et al. in [19] where an improved version version of A(k) indexes is proposed, called P(k) indexes, which for each k is shown to be minimal and precise for a certain well-defined practical XPath fragment.

Structural indexes combined with other indexing techniques Structural indexes can be combined and extended with other indexing techniques. For example, Kaushik et al. present in [38] an information retrieval system that retrieves XML documents based on path expression that may contain in some positions keywords. In order to efficiently match the keywords, the structural index is extended with an inverted list that maps keywords to element and text nodes, but also provides a pointer to the index node whose payload contains that element or text node. This can be seen as an extension

to the indexing mechanism of SIREn, as discussed in Section 1.3.2, but now the small trees that represent entities are generalized to XML documents.

Another combination is presented by Arion et al. in [3] where XML documents are indexed with a structural index that is essentially a 1-index as presented in Section 1.4.2 except that it is required to be a tree. Although this may make the index larger then necessary, this is compensated by the fact that it allows the use of the $(pre, post, level)$ labeling scheme on the index itself, which allows faster execution of path expressions over this tree structure.

Parameterizable Index Graphs One of the first instances where structural indexes were explicitly used for RDF graphs is by Tran and Ladwig [70]. In their work it is assumed that the RDF graph is essentially an edge-labeled graph with the edge labels explicitly distinct from the graph nodes. Consequently, in basic graph patterns it is not allowed to have variables in the predicate position of a triple pattern. Although this excludes certain typical RDF use cases where data and meta-data are queried together, it makes the indexing and query optimization problem more similar to that in classical graph databases.

The fundamental notion is that of *Parameterizable Index Graph*, or PIG, which is a structural index based on forward and backward bisimulation, as was introduced by Kaushik et al. in [37]. For a certain index, the types of edges that are considered for this bisimilarity can be restricted as specified by two sets: one containing the edges labels, i.e., predicate names, that are considered for the forward direction, and the other set those for the backward direction. These sets can be used to tune the index for certain workloads.

Based on the PIG three concrete indexing structures are defined: (1) An index we will call *PIGidx* that represents the PIG itself and maps predicate names to a sorted list of pairs that represent the edges in the PIG associated with that predicate name. (2) An index we will call *VPSOidx* over keyspace (v, p, s, o) with v a node in PIG, that allow prefix matching, i.e., we can do lookups for v, vp or vps, and returns sorted results. A tuple (v, p, s, o) is indexed if (s, p, o) is in the RDF graph and s is in the payload of v. (3) A similar index we will call *VPOSidx* over keyspace (v, p, o, s).

Given these indexes the result of a query that is a basic graph pattern can then be computed as follows. First we apply the query to the PIG and obtain answers in terms of PIG nodes. This can be done by iterating over the triple patterns in the basic graph pattern, and for each extend the intermediate result by equijoining it appropriately with the pairs associated by PIGidx with the mentioned predicate name. For a query with n triple patterns the result is a set of tuples of the form $T = (v_1, \ldots, v_{2n})$ where for $1 \leq i \leq n$ each pair (v_{2i-1}, v_{2i}) is a local matching for the ith triple pattern. For such pairs (v_{2i-1}, v_{2i}) we can define an associated set L_i of pairs at the RDF graph level such that L_i contains (r_{2i}, r_{2i-1}) iff r_{2i} and r_{2i-1} are in the payload of (v_{2i-1} and v_{2i}) respectively, and (r_{2i}, p_i, r_{2i-1}) is in the RDF graph, where

p_i the predicate in triple pattern i. For each such T we can then construct a set of corresponding tuples of $2n$ resources in the RDF graph by equijoining these L_i using appropriate equality conditions. Depending on the equality conditions and the join order we can compute each additional join step using either VPSOidx or VPOSidx plus some selections. Finally, we take the union of all these sets, and project the result such that we have only one column for each variable in the query.

There are several ways in which these proposed indexes and the query executing procedure can be optimized. The PIG can, for example, be made smaller by restricting the bisimulation to depth k as in $\mathrm{A}(k)$ and $\mathrm{P}(k)$ indexes. The query execution can be optimized by considering different join orders, and since for certain tree-shaped queries PIGidx is in fact a precise index, these can sometimes be pruned from a query after the first step of query execution over PIGidx.

gStore Another RDF datastore that features indexing structures that can be thought of as a structural indexes is gStore, presented by Zou et al. in [86]. In this system the focus is on answering queries with wildcards over disk-based data, and supporting this with an index that is easy to maintain. Also here, as for parameterized index graphs, the assumption is made that the RDF graph is treated as essentially an edge labeled graph.

The key idea of gStore is to assign to each resource not only an integer identifier but also a *signature*. The signature is a bit-string that contains some information about the triples in which the resource occurs as the subject. It is determined as follows. First, we define for each of these relevant triples a bit-string of $M + N$ bits, where M bits are used to encoding the predicate and N bits are used to encode the object. The predicate is encoded in M bits by applying some appropriate hash function. If the object is a resource then it is also mapped by means of a hash function to an N bits number. If the object is a literal, however, then we determine the N bits as follows: first determine the set of 3-grams of the literal, and then apply a hash function h that maps each 3-gram to a number in $[1..N]$. We set the ith bit to 1 iff there is a 3-gram in the set that is mapped by h to i. Given these encodings of each triple in $M + N$ bits we then combine all these bit-strings into a single bit-string of $M + N$ bits by taking their bitwise OR (remember that all of the triples have the same subject s). The key insight is that if we do the same for variables in a basic graph pattern, while mapping variables in the object to $M + N$ zeros, this gives us a quick way to filter out certain impossible matches by checking if the bit-string of the variable is subsumed by the bit-string of the resource.

The indexing structure built over the signatures of resources is called a VS-tree, a vertex signature tree. This is a balanced tree such that each of its leaves corresponds to a signature of a resource and the internal nodes correspond to signatures obtained by taking the bit-wise OR of the children's signatures. The tree is optimized to efficiently locate a certain signature of

a resource when starting from the root. At each level in the tree, we also add edges between tree nodes at the same level. Such an edge indicates the existence of at least one triple between resources associated with the leaves of the subtrees rooted at these tree nodes. More precisely, between index tree nodes m_1 and m_2 at level l there will be an edge iff there is a leaf node m_1' below m_1 and a leaf node m_2' below m_2 such that these correspond to signatures of the resources r_1 and r_2, respectively, and there is a triple of the form (r_1, p, r_2). Moreover, this edge between m_1 and m_2 in the index is labeled with a bit-string of M bits which is the bit-wise OR of the signatures of all predicates that hold between all such r_1 and r_2. Note that each higher level of this index we therefore find a smaller, more abstract and summarized structural index of the original RDF graph. This allows us to determine all matchings of a basic graph pattern, by starting from the root and moving one level down each time, each step eliminating more impossible matchings and refining those that remain, until we have matchings on the leaves. In the final step we can translate these to matchings on the actual resources which we can check against the actual RDF graph.

1.5 Conclusion and Future Work

In this chapter we have provided an up to date survey of the rich variety of approaches to storing and indexing very large RDF data sets. As we have seen this is a very active area of research, with sophisticated contributions from several fields. The survey has been organized by the three main perspectives which unify the many efforts across these fields: relational, entity, and graph-based views on RDF data.

We conclude with brief indications for further research. Across all perspectives, a major open issue is the incorporation of schema and ontology reasoning (e.g., RDFS and OWL) in storage and indexing. In particular, there has been relatively little work on the impact of reasoning on disk-based data structures (e.g., see [27, 69]). Furthermore, efficient maintenance of storage and indexing structures as data sets evolve is a challenge for all approaches where many research questions remain open. Possible directions for additional future work in the entity perspective are the investigation of support for richer query languages and integration of entity-centric storage and indexing with techniques from the other two perspectives. Finally, it is clear from the survey that graph-based storage and indexing is currently the least developed perspective. A major additional direction for research here is the development and study of richer structural indexing techniques and related query processing strategies, following the success of such approaches for semi-structured data.

Acknowledgements The research of FP is supported by a FNRS/FRIA scholarship. The research of SV is supported by the OSCB project funded by the Brussels Capital Region. The research of GF, JH, and YL is supported by the Netherlands Organisation for Scientific Research (NWO).

References

1. Abadi, D., Marcus, A., Madden, S., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for semantic web data management. The VLDB Journal **18**, 385–406 (2009)
2. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40**, 1:1–1:39 (2008)
3. Arion, A., Bonifati, A., Manolescu, I., Pugliese, A.: Path summaries and path partitioning in modern XML databases. World Wide Web **11**, 117–151 (2008)
4. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In: Proceedings of the 19th international conference on World Wide Web, WWW '10, pp. 41–50. ACM, New York, NY, USA (2010)
5. Bizer, C., Jentzsch, A., Cyganiak, R.: State of the LOD cloud. http://www4.wiwiss.fu-berlin.de/lodcloud/state/. Retrieved July 5, 2011
6. Bönström, V., Hinze, A., Schweppe, H.: Storing RDF as a graph. In: Proceedings of the First Conference on Latin American Web Congress, pp. 27–36. IEEE Computer Society, Washington, DC, USA (2003)
7. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: International Semantic Web Conference, pp. 54–68. Sardinia, Italy (2002)
8. Castillo, R.: RDFMatView: Indexing RDF data for SPARQL queries. In: 9th International Semantic Web Conference (ISWC2010) (2010)
9. Center, Q.G.: Bio2RDF. http://bio2rdf.org/
10. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Qeurying Scheme. In: VLDB, pp. 1216–1227. Trondheim, Norway (2005)
11. Data.gov. http://www.data.gov
12. Delbru, R., Campinas, S., Tummarello, G.: Searching web data: an entity retrieval and high-performance indexing model. Web Semantics: Science, Services and Agents on the World Wide Web **In Press, Accepted Manuscript**, – (2011)
13. Dong, X., Halevy, A.Y.: Indexing Dataspaces. In: ACM SIGMOD, pp. 43–54. Beijing (2007)
14. Elbassuoni, S., Ramanath, M., Schenkel, R., Weikum, G.: Searching RDF graphs with SPARQL and keywords. IEEE Data Eng. Bull. **33**(1), 16–24 (2010)
15. Erling, O.: Towards web scale RDF. In: SSWS. Karlsruhe, Germany (2008)
16. Erling, O., Mikhailov, I.: RDF support in the virtuoso DBMS. In: S. Auer, C. Bizer, C. Müller, A.V. Zhdanova (eds.) CSSW, *LNI*, vol. 113, pp. 59–68. GI (2007)
17. Fletcher, G.H.L., Beck, P.W.: Scalable indexing of RDF graphs for efficient join processing. In: CIKM, pp. 1513–1516. Hong Kong (2009)
18. Fletcher, G.H.L., Van Den Bussche, J., Van Gucht, D., Vansummeren, S.: Towards a theory of search queries. ACM Trans. Database Syst. **35**, 28:1–28:33 (2010)
19. Fletcher, G.H.L., Van Gucht, D., Wu, Y., Gyssens, M., Brenes, S., Paredaens, J.: A methodology for coupling fragments of XPath with structural indexes for XML documents. Information Systems **34**(7), 657–670 (2009)
20. Franklin, M., Halevy, A., Maier, D.: From databases to dataspaces: a new abstraction for information management. SIGMOD Rec. **34**, 27–33 (2005)

21. Furche, T., Weinzierl, A., Bry, F.: Labeling RDF graphs for linear time and space querying. In: R. De Virgilio, F. Giunchiglia, L. Tanca (eds.) Semantic Web Information Management, pp. 309–339. Springer (2009)
22. Goasdoué, F., Karanasos, K., Leblay, J., Manolescu, I.: Rdfviews: a storage tuning wizard for rdf applications. In: J. Huang, N. Koudas, G.J.F. Jones, X. Wu, K. Collins-Thompson, A. An (eds.) CIKM, pp. 1947–1948. ACM (2010)
23. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: VLDB, pp. 436–445. Athens, Greece (1997)
24. Gou, G., Chirkova, R.: Efficiently querying large XML data repositories: A survey. IEEE Trans. Knowl. Data Eng. **19**(10), 1381–1403 (2007)
25. Groppe, S., Groppe, J., Linnemann, V.: Using an Index of Precomputed Joins in Order to Speed up SPARQL Processing. In: ICEIS, pp. 13–20. Funchal, Madeira, Portugal (2007)
26. Gyssens, M., Paredaens, J., Van Gucht, D., Fletcher, G.H.L.: Structural Characterizations of the Semantics of XPath as Navigation Tool on a Document. In: ACM PODS, pp. 318–327. Chicago (2006)
27. Haffmans, W.: A study of efficient RDFS entailment in external memory. Master's thesis, Eindhoven University of Technology (2011)
28. Halevy, A.Y.: Answering queries using views: A survey. VLDB J. **10**(4), 270–294 (2001)
29. Halevy, A.Y., Franklin, M.J., Maier, D.: Principles of dataspace systems. In: PODS, pp. 1–9. Chicago (2006)
30. Harris, S.: SPARQL query processing with conventional relational database systems. Web Information Systems Engineering WISE 2005 Workshops **3807**, 235–244 (2005)
31. Harris, S., Gibbins, N.: 3store: Efficient bulk RDF storage. In: PSSS1, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, pp. 1–15. Sanibel Island, Florida (2003)
32. Harth, A., Decker, S.: Optimized index structures for querying RDF from the web. In: IEEE LA-WEB, pp. 71–80. Buenos Aires, Argentina (2005)
33. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: ISWC. Busan, Korea (2007)
34. Hayes, P.: RDF Semantics. W3C Recommendation (2004)
35. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool (2011)
36. Hertel, A., Broekstra, J., Stuckenschmidt, H.: RDF storage and retrieval systems. In: S. Staab, D. Rudi Studer (eds.) Handbook on Ontologies, International Handbooks on Information Systems, pp. 489–508. Springer Berlin Heidelberg (2009)
37. Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering Indexes for Branching Path Queries. In: ACM SIGMOD, pp. 133–144. Madison, WI (2002)
38. Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R.: On the Integration of Structure Indexes and Inverted Lists. In: ACM SIGMOD, pp. 779–790. Paris (2004)
39. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In: IEEE ICDE, pp. 129–140. San Jose, CA (2002)
40. Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation (2004)
41. Kolas, D., Emmons, I., Dean, M.: Efficient linked-list RDF indexing in parliament. In: A. Fokoue, Y. Guo, T. Liebig (eds.) Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009), *CEUR*, vol. 517, pp. 17–32. Washington DC, USA (2009)
42. Kolas, D., Self, T.: Spatially-augmented knowledgebase. In: K. Aberer, K.S. Choi, N.F. Noy, D. Allemang, K.I. Lee, L.J.B. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, P. Cudré-Mauroux (eds.) ISWC/ASWC, *Lecture Notes in Computer Science*, vol. 4825, pp. 792–801. Springer (2007)

43. Konstantinou, N., Spanos, D.E., Mitrou, N.: Ontology and database mapping: A survey of current implementations and future directions. J. Web Eng. **7**(1), 1–24 (2008)
44. Ladwig, G., Tran, T.: Combining query translation with query answering for efficient keyword search. In: ESWC, pp. 288–303. Crete (2010)
45. Lee, J., Pham, M.D., Lee, J., Han, W.S., Cho, H., Yu, H., Lee, J.H.: Processing SPARQL queries with regular expressions in RDF databases. In: Proceedings of the ACM fourth international workshop on Data and text mining in biomedical informatics, DTMBIO '10, pp. 23–30. ACM, New York, NY, USA (2010)
46. Levandoski, J.J., Mokbel, M.F.: RDF data-centric storage. In: ICWS, pp. 911–918. IEEE (2009)
47. Li, Q., Moon, B.: Indexing and querying xml data for regular path expressions. In: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, pp. 361–370. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
48. Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Agarwal, R.: Dynamic maintenance of web indexes using landmarks. In: Proceedings of the 12th international conference on World Wide Web, WWW '03, pp. 102–111. ACM, New York, NY, USA (2003)
49. Ma, L., Su, Z., Pan, Y., Zhang, L., Liu, T.: RStar: An RDF Storage and Query System for Enterprise Resource Management. In: ACM CIKM, pp. 484–491. Washington, D.C. (2004)
50. del Mar Roldán García, M., Montes, J.F.A.: A Survey on Disk Oriented Querying and Reasoning on the Semantic Web. In: IEEE ICDE Workshop SWDB. Atlanta (2006)
51. Matono, A., Amagasa, T., Yoshikawa, M., Uemura, S.: An indexing scheme for RDF and RDF schema based on suffix arrays. In: SWDB, pp. 151–168. Berlin (2003)
52. Matono, A., Amagasa, T., Yoshikawa, M., Uemura, S.: A path-based relational RDF database. In: ADC, pp. 95–103. Newcastle, Australia (2005)
53. McGlothlin, J.P., Khan, L.R.: Rdfkb: efficient support for rdf inference queries and knowledge management. In: B.C. Desai, D. Saccà, S. Greco (eds.) IDEAS, ACM International Conference Proceeding Series, pp. 259–266. ACM (2009)
54. Milo, T., Suciu, D.: Index Structures for Path Expressions. In: ICDT, pp. 277–295. Jerusalem (1999)
55. Minack, E., Sauermann, L., Grimnes, G., Fluit, C.: The sesame lucenesail: Rdf queries with full-text search. Tech. rep., NEPOMUK Consortium, (2008)
56. Neumann, T., Weikum, G.: RDF-3X: A RISC-style engine for RDF. In: VLDB. Auckland, New Zealand (2008)
57. Neumann, T., Weikum, G.: x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. Proc. VLDB Endow. **3**, 256–263 (2010)
58. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale semantic web data. J. Web Semantics **7**(4), 305 – 316 (2009)
59. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008)
60. Ramanan, P.: Covering indexes for XML queries: bisimulation - simulation = negation. In: Proceedings of the 29th international conference on Very large data bases - Volume 29, VLDB '2003, pp. 165–176. VLDB Endowment (2003)
61. Rao, P., Moon, B.: Sequencing XML data and query twigs for fast pattern matching. ACM Trans. Database Syst. **31**, 299–345 (2006)
62. Rohloff, K., Dean, M., Emmons, I., Ryder, D., Sumner, J.: An Evaluation of Triple-Store Technologies for Large Data Stores. In: OTM 2007 Workshop SSWS, pp. 1105–1114. Vilamoura, Portugal (2007)
63. Rohloff, K., Schantz, R.E.: Clause-iteration with mapreduce to scalably query datagraphs in the shard graph-store. In: Proceedings of the fourth international workshop on Data-intensive distributed computing, DIDC '11, pp. 35–44. ACM, New York, NY, USA (2011)

64. Sakr, S., Al-Naymat, G.: Relational processing of RDF queries: A survey. ACM SIGMOD Record **38**, 23–28 (2009). URL http://www.sigmod.org/publications/sigmod-record/0912
65. Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., Pinkel, C.: An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In: ISWC, pp. 82–97. Karlsruhe (2008)
66. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^{2}Bench: A SPARQL Performance Benchmark. In: IEEE ICDE. Shanghai (2009)
67. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S.: Column-store support for RDF data management: not all swans are white. Proc. VLDB Endow. **1**, 1553–1563 (2008)
68. Sintek, M., Kiesel, M.: RDFBroker: A signature-based high-performance RDF store. In: ESWC, pp. 363–377. Budva, Montenegro (2006)
69. Theoharis, Y., Christophides, V., Karvounarakis, G.: Benchmarking database representations of RDF/S stores. In: Y. Gil, E. Motta, V.R. Benjamins, M.A. Musen (eds.) International Semantic Web Conference, *Lecture Notes in Computer Science*, vol. 3729, pp. 685–701. Springer (2005)
70. Tran, T., Ladwig, G.: Structure index for RDF data. In: Workshop on Semantic Data Management (SemData@ VLDB) (2010)
71. Tran, T., Wang, H., Rudolph, S., Cimiano, P.: Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In: ICDE, pp. 405–416. Shanghai (2009)
72. Tummarello, G., Cyganiak, R., Catasta, M., Danielczyk, S., Delbru, R., Decker, S.: Sig.ma: Live views on the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web **8**(4), 355–364 (2010). DOI 10.1016/j.websem.2010.08.003
73. Udrea, O., Pugliese, A., Subrahmanian, V.S.: GRIN: A graph based RDF index. In: AAAI, pp. 1465–1470. Vancouver, B.C. (2007)
74. Valduriez, P.: Join indices. ACM Trans. Database Syst. **12**, 218–246 (1987)
75. W3C SWEO Community Project: Linking open data. http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData
76. Wang, H., Liu, Q., Penin, T., Fu, L., Zhang, L., Tran, T., Yu, Y., Pan, Y.: Semplore: A scalable IR approach to search the Web of Data. Web Semantics: Science, Services and Agents on the World Wide Web **7**(3), 177–188 (2009)
77. Wang, X., Wang, S., Pufeng, D., Zhiyong, F.: Path summaries and path partitioning in modern XML databases. International Journal of Modern Education and Computer Science **3**, 55–61 (2011)
78. Weikum, G., Theobald, M.: From information to knowledge: harvesting entities and relationships from web sources. In: PODS, pp. 65–76. Indianapolis (2010)
79. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. In: VLDB. Auckland, New Zealand (2008)
80. Wilkinson, K.: Jena Property Table Implementation. In: SSWS, pp. 35–46. Athens, Georgia, USA (2006)
81. Witten, I., Moffat, A., Bell, T.: Managing gigabytes: compressing and indexing documents and images. Morgan Kaufmann (1999)
82. Wood, D., Gearon, P., Adams, T.: Kowari: A Platform for Semantic Web Storage and Analysis. In: XTech. Amsterdam (2005)
83. Wu, G., Li, J.: Managing large scale native RDF semantic repository from the graph model perspective. In: ACM SIGMOD Workshop IDAR, pp. 85–86. Beijing (2007)
84. Wu, Y., Gucht, D.V., Gyssens, M., Paredaens, J.: A study of a positive fragment of path queries: Expressiveness, normal form and minimization. Comput. J. **54**(7), 1091–1118 (2011)
85. Yang, M., Wu, G.: Caching intermediate result of sparql queries. In: S. Srinivasan, K. Ramamritham, A. Kumar, M.P. Ravindra, E. Bertino, R. Kumar (eds.) WWW (Companion Volume), pp. 159–160. ACM (2011)
86. Zou, L., Mo, J., Chen, L., Özsu, M.T., Zhao, D.: gStore: Answering SPARQL queries via subgraph matching. Proceedings of the VLDB Endowment **4**(8), 482–493 (2011)