Database foundations for scalable RDF processing

Katja Hose[1], Ralf Schenkel[1,2], Martin Theobald[1], and Gerhard Weikum[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Saarland University, Saarbrücken, Germany

Abstract. As more and more data is provided in RDF format, storing huge amounts of RDF data and efficiently processing queries on such data is becoming increasingly important. The first part of the lecture will introduce state-of-the-art techniques for scalably storing and querying RDF with relational systems, including alternatives for storing RDF, efficient index structures, and query optimization techniques. As centralized RDF repositories have limitations in scalability and failure tolerance, decentralized architectures have been proposed. The second part of the lecture will highlight system architectures and strategies for distributed RDF processing. We cover search engines as well as federated query processing, highlight differences to classic federated database systems, and discuss efficient techniques for distributed query processing in general and for RDF data in particular. Moreover, for the last part of this chapter, we argue that extracting knowledge from the Web is an excellent showcase – and potentially one of the biggest challenges – for the scalable management of uncertain data we have seen so far. The third part of the lecture is thus intended to provide a close-up on current approaches and platforms to make reasoning (e.g., in the form of probabilistic inference) with uncertain RDF data scalable to billions of triples.

1 RDF in centralized relational databases

The increasing availability and use of RDF-based information in the last decade has led to an increasing need for systems that can store RDF and, more importantly, efficiency evaluate complex queries over large bodies of RDF data. The database community has developed a large number of systems to satisfy this need, partly reusing and adapting well-established techniques from relational databases [122]. The majority of these systems can be grouped into one of the following three classes:

1. *Triple stores* that store RDF triples in a single relational table, usually with additional indexes and statistics,
2. *vertically partitioned tables* that maintain one table for each property, and
3. Schema-specific solutions that store RDF in a number of *property tables* where several properties are jointly represented.

In the following sections, we will describe each of these classes in detail, focusing on two important aspects of these systems: *storage and indexing*, i.e., how are RDF triples mapped to relational tables and which additional support structures are created; and *query processing*, i.e., how SPARQL queries are mapped to SQL, which additional operators are introduced, and how efficient execution plans for queries are determined. In addition to these purely relational solutions, a number of specialized RDF systems has been proposed that built on non-relational technologies, we will briefly discuss some of these systems. Note that we will focus on SPARQL[3] processing, which is not aware of underlying RDF/S or OWL schema and cannot exploit any information about subclasses; this is usually done in an additional layer on top.

We will explain especially the different storage variants with the running example from Figure 1, some simple RDF *facts* from a university scenario. Here, each line corresponds to a fact (*triple*, *statement*), with a *subject* (usually a resource), a *property* (or *predicate*), and an *object* (which can be a resource or a constant). Even though resources are represented by URIs in RDF, we use string constants here for simplicity. A collection of RDF facts can also be represented as a graph. Here, resources (and constants) are nodes, and for each fact `<s,p,o>`, an edge from s to o is added with label p. Figure 2 shows the graph representation for the RDF example from Figure 1.

```
<Katja,teaches,Databases>
<Katja,works_for,MPI Informatics>
<Katja,PhD_from,TU Ilmenau>
<Martin,teaches,Databases>
<Martin,works_for,MPI Informatics>
<Martin,PhD_from,Saarland University>
<Ralf,teaches,Information Retrieval>
<Ralf,PhD_from,Saarland University>
<Ralf,works_for,Saarland University>
<Saarland University,located_in,Germany>
<MPI Informatics,located_in,Germany>
```

Fig. 1. Running example for RDF data

1.1 Triple Stores

Triple stores keep RDF triples in a simple relational table with three or four attributes. This very generic solution with low implementation overhead has been very popular, and a large number of systems based on this principle are available. Prominent examples include 3store [56] and Virtuoso [41] from the Semantic Web community, and RDF-3X [101] and HexaStore [155] that were developed by database groups.

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

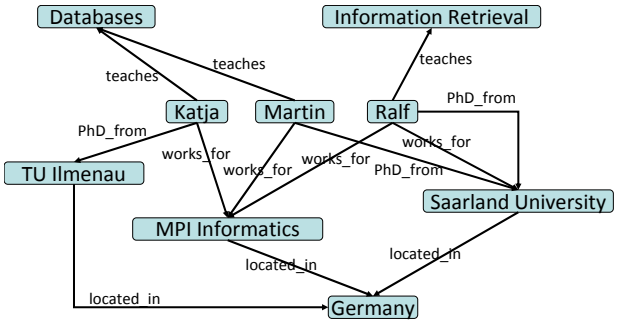

Fig. 2. Graph representation for the RDF example from Figure 1

Storage. RDF facts are mapped to a generic three-attribute table of the form `(subject,property,object)`, also know as *triple table*; for simplicity, we will abbreviate the attributes by `S`, `P`, and `O`. To save space (and to make access structures more efficient), most systems convert resource identifiers, properties and constants to numeric ids before storing them in the relation, for example by hashing. The resulting map is usually stored in an additional table, sometimes separately for resource ids and constants. If a system stores data from more than one source (or more than one RDF graph), the relation is often extended by a fourth numeric attribute, the `graph id` (abbreviated as `G`), that uniquely identifies the source of a triple. In this case, the relation is also called a *quadruple table*.

Figure 3 shows the resulting three-attribute relation for the example from Figure 1.

For efficient query processing, indexes on (a subset of) all combinations of `S`, `P`, and `OS` are maintained. This allows to efficiently retrieve all matches for a triple pattern of a SPARQL query. We will often refer to indexes with the sequence of the abbreviations of the indexed attributes (such as `SPO`). Since each index has approximately the size of the relation, the number of combinations for which indexes are kept is usually limited, or indexes are stored in a compressed way.

Virtuoso [40] comes with a space-optimized way of mapping resources, predicates and constants to numeric ids (IRI ID). These strings are mapped to numeric ids only if they are long (which means at least 9 bytes long), otherwise, they are stored as text in the quadruple relation (this saves for short objects over a solu-

subject	property	object
Katja	teaches	Databases
Katja	works_for	MPI Informatics
Katja	PhD_from	TU Ilmenau
Martin	teaches	Databases
Martin	works_for	MPI Informatics
Martin	PhD_from	Saarland University
Ralf	teaches	Information Retrieval
Ralf	PhD_from	Saarland University
Ralf	works_for	Saarland University
Ralf	works_for	MPI Informatics
Saarland University	locatedIn	Germany
MPI Informatics	located_in	Germany

Fig. 3. Triple store representation for the running example from Figure 1

tion that maps everything to ids). It uses a standard quadruple table `(G,S,P,O)` with a primary key index on all four attributes together. In addition, it uses a bitmap index on `(O,G,P,S)`: This index maintains, for each combination of `(O,G,P)` in the data, a bit vector. Each subject is assigned a bit position, and for a quadruple (g,s,p,o) in the data, the bit position for s is set to 1 in the bit vector for (o,g,p). Virtuoso stores distinct values within a page only once and eliminates common prefixes of strings. An additional compression of each page with gzip yields a compression from 8K to 3K for most pages.

RDF-3X uses a standard triple table and does not explicitly support multiple graphs. However, RDF-3X never actually materializes this table, but instead keeps clustered B+ tree indexes on all six combinations of (S,P,O). Additionally, RDF-3X includes aggregated indexes for each possible pair of (S,P,O) and each order, resulting in six additional indexes. The index on (S,O), for example, stores for each pair of subject and object that occurs in the data the number of triples with this subject and this object, we'll refer to this index as `SO*`. Such an index allows to efficiently answer queries like `select ?s ?o where {?s ?p ?o}`. We could process this by scanning the `SOP` index, but we don't need the exact bindings for `?p` to generate the result, so we read many index entries that don't add new results. All we need is, for each binding of `?s` and `?o`, the number of triples with this subject and this object, so that we can generate the right number of results for this binding (including duplicates). The `SO*` index can help a lot here. Finally, RDF-3X maintains aggregated indexes for each single attribute, again plus triple counts.

To reduce space requirements of these indexes, RDF-3X stores the leaves of the indexes in pages and compresses them. Since these leaves contain triples that often share some attribute values and all attributes are numeric, it uses delta encoding for compression. This, together with encoding strings into comparably short numbers, helps to keep the overall size of the database comparable or even slightly smaller than the size of the uncompressed RDF triples in textual representation. The original RDF-3X paper [101] includes a discussion of space-

time tradeoffs for compression and shows that, for example, compression with LZ77 generates more compact indexes, but requires significantly more time to decompress.

Query Processing and Optimization. Query execution on a quadruple store is done in two steps, *converting* the SPARQL query into an equivalent SQL query, and creating and executing a *query plan* for this SQL query.

Step 1. The conversion of a SPARQL query to an equivalent SQL query on the triple/quadruple table is a rather straight-forward process; we'll explain it now for triple tables. For each triple pattern in the SPARQL query, a copy of the triple relation is added to the query. Whenever a common variable is used in two patterns, a join between the corresponding relation instances is created on the attributes where the variable occurs. Any constants are directly mapped to conditions on the corresponding relation's attribute.

As an example, consider the SPARQL query

```
SELECT ?a ?b WHERE
   {?a works_for ?u.
    ?b works_for ?u.
    ?a phd_from ?u. }
```

which selects people who work at the same place where they got their phd, together with their coworkers. This is mapped to the SQL query

```
SELECT t1.s, t2.s FROM triple t1, triple t2, triple t3
WHERE t1.p='works_for'
  AND t2.p='works_for'
  AND t3.p='phd_from'
  AND t1.o=t2.o
  AND t1.o=t3.o
  AND t1.s=t3.s
```

Note that in a real system, the string constants would usually be mapped to the numeric id space first. Further note that we can optimize away one join here (`t2.o=t3.o`) since it is redundant.

Step 2. Now that we have a standard SQL query, it is tempting to simply rely on the existing relational backends for optimizing and processing this query. This is actually done in many systems, and even those systems which implement their own backend system use the same operators used in relational databases. Converting the SQL query into an equivalent abstract operator tree, for example an expression in relational algebra, is again straight-foward.

Once this is done, the next step is creating an efficient physical execution plan, i.e., decide how the abstract operators (joins, projection, selection) are mapped to physical implementations in a way that the resulting execution is as cheap (in terms of I/O and CPU usage) and as fast (in terms of processing times) as possible. The choice of implementations is rather huge, for example a join operator can be implemented with merge joins, hash joins, or nested loop

joins. Additionally, a number of specialized joins exist (such as outer joins and semi joins) that can further improve efficiency. An important physical operator in many systems are index lookups and scans which exploit the numerous indexes that systems keep. Often, each triple pattern in the original SPARQL query corresponds to an index scan in the corresponding index if that index is available, for example, the triple pattern `?a works_for ?b` could be mapped on a scan of the `PSO` index, if that exists. If the optimal index does not exist, scans of less specific indexes can be used, but some information from that index must be skipped. For example, if the system provides only an `O` index, a pattern `?a works_for MPI Informatics` can be mapped to a scan of the `O` index, starting at `MPI Informatics` and skipping all entries that do not match the predicate constraint.

Finding the most efficient plan now includes considering possible variants of physical plans (such as different join implementations, different join orders, etc.) and selecting the most efficient plan. This, in turn, requires that the execution cost for each plan is estimated. It turns out that off-the-shelf techniques implemented in current relational databases, for example attribute-level histograms to represent the distribution. These techniques were not built for dealing with a single, large table. The main problem is that they ignore correllation of attributes, since statistics are available only separately for each attribute. Estimates (for example how many results a join will have, or how many results a selection will have) are therefore often way off, which can lead to arbitrarily bad execution plans. Multi-dimensional histograms, on the other hand, could capture this correllation, but can easily get too large for large-scale RDF data.

RDF-3X [100, 101] comes with specializes data structures for maintaining statistics. It uses histograms that can handle any triple pattern and any join, but assume independence of different patterns, and it comes with optimizations for frequent join paths. To further speed up processing, it applies sideway information passing between operators [100]. It also includes techniques to deal with unselective queries which return a large fraction of the database.

Virtuoso [40] exploits the bit vectors in its indexes for simple joins, which can be expressed as a conjunction of thse sparse bit vectors. As an example, consider the SPARQL query

```
select ?a
where {?a works_for Saarland University.
       ?a works\for MPI Informatics.}
```

To execute this, it is sufficient to load the bit vectors for both triple patterns and intersect them. For cost estimation, Virtuoso does not rely on per-attribute histogram, but uses query-time sampling: If a triple pattern has constants for `p` and `o` and the graph is fixed, it loads the first page of the bit vector for that pattern from the index, and extrapolates selectivity from the selectivity of this small sample.

Further solutions on the problem of selectivity estimation for graph queries were proposed by [90, 91] outside the context of an RDF system; Stocker et al. [135] consider the problem of query optimization with graph patterns.

1.2 Vertically Partitioned Tables

The vast majority of triple patterns in queries from real applications has fixed properties. To exploit this fact for storing RDF, one table with two attributes, one for storing subjects and one for storing objects, is created for each property in the data; if quadruples are stored, a third attribute for the graphid is added. An RDF triple is now stored in the table for its property. Like in the Triple table solution, string literals are usually encoded as numeric ids. Figure 4 shows how our example data from Figure 1 is represented with vertically partitioned tables.

teaches	
subject	**object**
Katja	Databases
Martin	Databases
Ralf	Information Retrieval

works_for	
subject	**object**
Katja	MPI Informatics
Martin	MPI Informatics
Ralf	MPI Informatics
Ralf	Saarland University

PhD_from	
subject	**object**
Katja	TU Ilmenau
Martin	Saarland University
Ralf	Saarland University

located_in	
subject	**object**
Saarland University	Germany
MPI Informatics	Germany

Fig. 4. Representation of the running example from Figure 1 with vertically partitioned tables

Sinces tables have only two columns, this idea can be further pushed by not storing them in a traditional relational system (a *row store*), but in a *column store*. A column store does not store tables as collections of rows, but as collection of columns, where each entry of a column comes with a unique ID that allows to reconstruct the rows at query time. This has the great advantage that all entries within a column have the same type and can therefore be compressed very efficiently. The idea of using column stores for RDF was initially proposed by Abadi et al. [2], Sidirourgos et al. [130] pointed out advantages and disadvantages of this technique.

Regarding query processing, it is evident that triple patterns with a fixed property can be evaluated very efficiently, by simply scanning the table for this property (or, in a column store, accessing the columns of this table). Query optimization can also be easier as per-table statistics can be maintained. On the other hand, triple patterns with a property wildcard are very expensive since they need to access all two-column tables and form the union of the results.

1.3 Property Tables

In many RDF data collections, a large number of subjects have the same or at least are largely overlapping set of properties, and many of these properties will

be accessed together in queries (like in our example above that asked for people that did their PhD at the same place where they are working now). Combining all properties of a subject in the same table makes processing such queries much faster since there is no need for a join to combine the different properties. Property tables do exactly this: Groups of subjects with similar properties are represented by a single table where each attribute corresponds to a property. A set of facts for one of these subjects is then stored as one row in that table, where one column represents the subjects, and the other columns store objects for the properties that correspond to that column, or NULL if no such property exists. The most prominent example for this storage structure is Jena [23, 158]. Chong et al. [27] proposed property tables as external view (which can be materialized) to simplify access to triple stores. Levandoski et al [80] demonstrate that property tables can outperform triple stores and vertically partitioned tables for RDF data collections with regular structure, such as DBLP or DBPedia.

This table layout comes with two problems to solve: First, there should not be too many NULL values since they increase storage space, so storing the whole set of facts in a single table is not a viable solution. Instead, the set of subjects to store in one table can be determined by clustering subjects by the set of their properties, or subjects of the same type can be stored in the same table if schema information is available. Second, multi-valued properties, i.e., properties that can have more than one object for the same subject, cannot be stored in this way without breaking the relational paradigm. In our example, people can work for more than one institution at the same time. To solve this, one can either create multiple attributes for the same property, but this works only if the maximal number of different objects for the same property and the same subject is rather small. Alternatively, one can store facts with these properties in a standard triple table.

Figure 5 shows the representation of the example from Figure 1 with property tables. We grouped information about people in the `People` table and information about Institutions in the `Institutions` table. As the `works_for` property can have multiple objects per subject, we store facts with this property in the `Remainder` triple table.

1.4 Specialized Systems

Beyond the three classes of systems we presented so far, there are a number of systems that don't fit into these categories. We will shortly sketch these systems here without giving much detail, and refer the reader to the referenced original papers for more information.

Atre et al. [9] recently proposed to store RDF data in matrix form. Compressed bit vectors help to make query processing efficient. Zhou and Wu [160] propose to split RDF data into XML trees, rewrite SPARQL as XPath and XQuery expressions, and implement an RDF system on top of an XML database. Fletcher and Beck [42] propose to index not triples, but atoms, and introduce the Three-Way Triple Tree, a disk-based index.

People		
subject	**teaches**	**PhD_from**
Katja	Databases	TU Ilmenau
Martin	Databases	Saarland University
Ralf	Information Retrieval	Saarland University

Institutions	
subject	**located_in**
Martin	Saarland University
Ralf	Saarland University

Remainder		
subject	**predicate**	**object**
Katja	works_for	MPI Informatics
Martin	works_for	MPI Informatics
Ralf	works_for	Saarland University
Ralf	works_for	MPI Informatics

Fig. 5. Representation of the running example from Figure 1 with property tables

A number of proposals aims at representing and indexing RDF as graphs. Baolin and Bo [85] combine in their HPRD system a triple index with a path index and a content index. Liu and Hu [84] propose to use a dedicated path index to improve efficiency of RDF query processing. Grin [149] explicitly uses a graph index for RDF processing. Matono et al. [92] propose to use a path index based on suffix arrays. Bröcheler et al propose to use DOGMA, a disk-based graph index, for RDF processing.

2 RDF in distributed setups

Managing and querying RDF data efficiently in a centralized setup is an important issue. However, with the ever-growing amount of RDF data published on the Web, we also have to pay attention to relationships between web-accessible knowledge bases. Such relationships arise when knowledge bases store semantically similar data that overlaps. For example, a source storing extracted information from Wikipedia (DBpedia [10]) and a source providing geographical information about places (GeoNames[4]) might both provide information about the same city, e.g., Berlin.

Such relationships are often expressed explicitly in the form of RDF links, i.e., subject and object URIs refer to different namespaces and therefore establish a semantic connection between entities contained in different knowledge bases. Consider again the above mentioned sources (DBpedia and GeoNames) [15], both provide information about the same entity (e.g. Berlin) but use different identifiers. Thus, the following RDF triple links the respective URIs and expresses that both sources refer to the same entity: (`<http://dbpedia.org/resource/Berlin>`, `<owl:sameAs>`, `http://sws.geonames.org/2950159/`)

The exploitation of these relationships and links offers users the possibility to obtain a wide variety of query answers that could not be computed considering

[4] `http://www.geonames.org/ontology/`

a single knowledge base but require the combination of knowledge provided by multiple sources. Computing such query answers requires sophisticated reasoning and query optimization techniques that also take the distribution into account. An important issue in this context that needs to be considered is the interface that is available to access a knowledge base [63]; some sources provide SPARQL endpoints that can answer SPARQL queries, whereas other sources offer RDF dumps – an RDF dump corresponds to a large RDF document containing the RDF graph representing a source's complete dataset.

The *Linking Open Data initiative*[5] is trying to enforce the process of establishing links between web-accessible RDF data sources – resulting in *Linked Open Data* [15], a detailed introduction to Linked Data is provided in [11]. The Linked Open Data cloud resembles the structure of the World Wide Web (web pages connected via hyperlinks) and relies on the so-called Linked Data principles [14]:

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
- Include links to other URIs, so that they can discover more things.

For a user there are several ways to access knowledge bases and exploit links between them. A basic solution is *browsing* [38,57,148]: the user begins with one data source and progressively traverses the Web by following RDF links to other sources. Browsers allow users to navigate through the sources and are therefore well-suited for non-experts. However, for complex information needs, formulating queries in a structured query language and executing them on the RDF sources is much more efficient. Thus, in the remainder of this section we will discuss the main approaches for distributed query processing on RDF data sources – a topic, which has also been considered in recent surveys [51,63]. We begin with an overview of search engines in Section 2.1 and proceed with approaches adhering to the principles of data warehouses (Section 2.2) and federated systems (Section 2.3). We proceed with approaches that discover new sources during query processing (Section 2.4) and end with approaches applying the P2P principle (Section 2.5).

2.1 Search engines

Before discussing how traditional approaches known from distributed database systems can be applied in the RDF and Linked Data context, let us discuss an alternative to find linked data on the Web: search engines.

The main idea is to crawl RDF data from the Web and create a centralized index based on the crawled data. The original data is not stored permanently but dropped once the index is created. Since the data provided by the original sources changes, indexes need to be recreated from time to time. In order to

[5] `http://esw.w3.org/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`

answer a user query, all that needs to be done is to perform a lookup operation on the index and to determine the set of relevant sources. In most cases user queries consist of keywords that the search engine tries to find matching data for. The so found results and relevant sources are output to the user with some additional information about the results.

The literature proposes several central index search engines, some of them are discussed in the following. We can distinguish local-copy approaches [26, 34, 58] that collect local copies of the crawled data and index-only approaches [37, 108] that only hold local indexes for the data found on the Web. Another distinguishing characteristic is what search engines find: RDF documents [37, 108] or RDF objects/entities [26, 34, 58].

Swoogle. One of the most popular search engines for the Semantic Web was Swoogle [37]. It was designed as a system to automatically discover Semantic Web documents, index their metadata, and answer queries about them. Using this engine, users can find ontologies, documents, terms, and other data published on the Web.

Swoogle uses an SQL database to store metadata about the documents, i.e., information about encoding, language, statistics, ontology annotations, relationships among documents (e.g., one ontology imports another one), etc. A crawler-based approach is used to discover RDF documents, metadata is extracted, and relationships between documents are computed. Swoogle uses two kinds of crawlers: a Google crawler using the Google web service to find relevant Semantic Web documents and a crawler based on Jena2 [157], which identifies Semantic Web content in a document, analyzes the content, and discovers new documents through semantic relations.

Discovered documents are indexed by an information retrieval system, which can use either character N-Gram or URIrefs as keywords to find relevant documents and to compute the similarity among a set of documents. Swoogle computes ranks for documents in a similar way as the PageRank [109] algorithm used by Google.

Thus, a user can formulate a query using keywords and Swoogle will report back a list of documents matching those keywords in a ranked order. Optionally, a user might also define content-based constraints to a general SQL query on the underlying database, e.g., the type of the document, the number of defined classes, language, encoding, etc.

Semantic Web Search Engine (SWSE). In contrast to other search engines (document-centric), which given a keyword look for relevant sources and documents, SWSE [58] is entity-centric, i.e., given a keyword it looks for relevant entities.

SWSE uses a hybrid approach towards data integration: first, it uses YARS2 [60] as an internal RDF store to manage the crawled data (data warehousing) and second, it applies virtual integration using wrappers for external sources. In order to increase linkage between data from different sources, SWSE

applies entity consolidation. The goal is to identify URIs that actually refer to the same real-world entity by finding matches analyzing values of inverse functional properties. In addition, existing RDF entities are linked to Web documents (e.g., HTML) using an inverted index over the text of documents and specific properties such as `foaf:name` to identify entities.

YARS2 uses three different index types to index the data: (i) a keyword index using Apache Lucene[6] as an inverted index – this index maps terms occurring in an RDF object of a triple to the subject, (ii) quad[7] indexes in six different orderings of triple components – distributed over a set of machines according to a hash function, and (iii) join indexes to speed up queries containing combinations of values or paths in the graph.

In the first step of searching, the user defines a keyword query. The result of the search is a list of all entities matching the keyword together with a summary description for the entities. The results are ranked by an algorithm similar to PageRank combining ranks from the RDF graph with ranks from the data source graph [65]. To refine the search and filter results, the user can specify a specific type/class, e.g., person, document, etc. By choosing a specific result entity, the user can obtain additional information. These additional pieces of information might originate from different sources. Users can then continue their exploration by following semantic links that might also lead to documents related to the queried entity.

WATSON. WATSON [34] is a tool and an infrastructure that automatically collects, analyzes, and indexes ontologies and semantic data available online in order to provide efficient access to this knowledge for Semantic Web users and applications.

The first step, of course, is crawling. WATSON tries to discover locations of semantic documents and collects them when found. The crawling process exploits well-known search engines and libraries, such as Swoogle and Google. The retrieved ontologies are inspected for links to other ontologies, e.g., exploiting `owl:import`, `rdfs:seeAlso`, dereferenceable URIs, etc.

Afterwards, the semantic content is validated, indexed, and metadata is generated. WATSON collects metadata such as the language, information about contained entities (classes, properties, literals), etc. By exploiting semantic relations (e.g. `owl:import`), implicit links between ontologies can be computed in order to detect and remove duplicate information and so that storing redundant information and presenting duplicated results to the user can be avoided.

WATSON supports queries based on keywords similar to Swoogle to retrieve and access semantic content including a particular search phrase (multiple keywords are supported). Keywords are matched against the local names, labels, comments and/or literals occurring in ontologies. WATSON returns URIs of matching entities, which can serve as an entry point for iterative search and exploration.

[6] `http://lucene.apache.org/java/docs/fileformats.html`
[7] A quad is a triple extended by a fourth value indicating the origin.

Sindice. Sindice [108] crawls RDF documents (files and SPARQL endpoints) from the Semantic Web and uses three indexes for resource URIs, Inverse Functional Properties (IFPs)[8], and keywords. Consequently, the user interface allows users to search for documents based on keywords, URIs, or IFPs. To process the query, the query only needs to be passed to the relevant index, results need to be gathered, and the output (HTML page) needs to be created.

The indexes correspond to inverted indexes of occurrences in documents. Thus, the URI index has one entry for each URI. The entry contains a list of document URLs that mention the corresponding URI. The structure of the keyword index is the same, the only difference is that it does not consider URIs but tokens (extracted from literals in the documents), the IFP index uses the uniquely identifying pair (property, value).

When looking for the URI of a specific entity (e.g. Berlin), Sindice provides the user with several documents that mention the searched URI. For each result some further information is given (human description, date of last update) to enable users to choose the best suitable source. The results are ranked in order of relevance, which is determined based on the TF/IDF relevance metric [43] in information retrieval, i.e., sources that share rare terms (URIs, IFPs, keywords) are preferred. In addition, the ranking prefers sources whose URLs a similar to queried URIs, i.e., containing the same hostnames.

Falcons. In contrast to other search engines, Falcons [26] is a keyword-based search engine that focuses on finding linked objects instead of RDF documents or ontologies. In this sense, it is similar to SWSE.

Just like other systems, Falcons crawls RDF documents, parses them using Jena[9], and follows URIs discovered in the documents for further crawling. To store the RDF triples, Falcons creates quadruples and uses a quadruple store (MySQL).

To provide detailed information about objects, the system constructs a virtual document containing literals associated with the object, e.g., human-readable names and descriptions (`rdfs:label`, `rdfs:comment`).

Falcons creates an inverted index based on the terms in the virtual documents and uses this index later on for keyword-based search. A second inverted index is built based on the objects' classes – it is used to perform filtering based on the objects' classes/types. For a query containing both, keywords and class restrictions, Falcons computes the intersection between the result sets returned by both indexes.

Thus, given a keyword query, Falcons uses the index to find virtual documents (and therefore objects) containing the keywords. Falcons supports class-based (typing) query refinement by employing class-inclusion reasoning, which is the main difference to SWSE, which does not allow such refinements and reasoning.

[8] `http://www.w3.org/TR/owl-ref/#InverseFunctionalProperty-def`: If a property is declared to be inverse-functional, then the object of a property statement uniquely determines the subject (some individual).

[9] `http://jena.sourceforge.net`

The so-found result objects are ranked by considering both their relevance to the query (similarity between the virtual documents and the keyword query) and their popularity (a measure based on the number of documents referring to the object).

Each presented result object is accompanied with a snippet that shows associated literals and linked objects matched with the query. – also shows detailed RDF descriptions loaded from the quadruple store.

More systems. There are many more search engines such as OntoSearch2 [111], which stores a copy of an ontology in a tractable description logic and supports SPARQL as a query language to find, for instance, all the instances of a given class or the relations occurring between two instances. Several other systems aim at providing efficient access to ontologies and semantic data available online. For example, OntoKhoj [112] is an ontology portal that crawls, classifies, ranks, and searches ontologies. For ranking they use an algorithm similar to PageRank. Oyster [110] is different in the sense that it focuses on ontology sharing: users register ontologies and their metadata, which can afterwards accessed over a peer-to-peer network of local registries. Finally let us mention OntoSelect [19] is an ontology library that focuses on providing natural language based access to ontologies.

2.2 Data warehousing

So far, we have discussed several techniques to search for documents and entities using search engines. However, the problem of answering queries based on data provided by multiple sources has been known in the database community since the 80s. Thus, from a database point of view the scenario we face with distributed RDF processing is similar to *data integration*.

Figure 6 shows a categorization of data integration systems from a classical database point of view. The same categorization can be applied to the problem of processing Linked Data because, despite some differences, similar problems have to be solved and similar techniques have already been applied by different systems for Linked Data processing.

The first and most important characteristic that distinguishes data integration systems is whether they copy the data into a central database, or storage system respectively, or leave the data at the sources and only work with statistical information about the sources, e.g., indexes. Approaches falling into the former category are generally referred to as *materialized data integration systems* with *data warehouses* as a typical implementation. Approaches of the second category are referred to as *virtual data integration systems* as they only virtually integrate the data without making copies.

The process of integrating data into a data warehouse is referred to as the ETL process (Extract-Transform-Load). First, the data is extracted from the sources, then it is transformed, and finally loaded into the data warehouse. This workflow can also be applied in the Semantic Web context.

Extract. As many knowledge bases are available as dumps for download, it is possible to download a collection of interesting linked datasets and import them into a data warehouse. This warehouse resembles a centralized RDF storage system that can be queried and optimized using the techniques discussed in Section 1. If a data source is not available for download, the data can be crawled by looking up URIs or accessing a SPARQL endpoint. As the data originates from different sources, the system should keep track of provenance, e.g., by using named graphs [22] or quads.

Transform. Data warehouses in the database world usually provide the data in an aggregated format, e.g., in a data warehouse storing information about sales we do not keep detailed information about all transactions (as provided by the sources) but, for instance, the volume of sales per day (computing aggregated values based on the data provided by the sources). For linked data, this roughly corresponds to running additional analyses on the crawled data for duplicate detection/removal or entity consolidation as applied, for instance, by WATSON and SWSE.

Load. Finally, the transformed data is loaded into the data warehouse. In dependence on aspects such as the update frequency of the sources and the size of the imported datasets, it might be difficult to keep the data warehouse up-to-date. In any case the data needs to be reloaded or recrawled so that the data warehouse can be updated accordingly. It is up to the user, or the user application respectively, to decide if and to what extent out-of-date data is acceptable or not.

Reconsidering the search engines we have already discussed above, we see that some of these approaches actually fall into this category: the search engines using a central repository to store a local copy of the RDF data they crawled from the web, e.g., SWSE [58], WATSON [34], and Falcons [26].

In summary, the data warehouse approach has some advantages and disadvantes. The biggest advantage, of course, is that all information is available locally, which allows for efficient query processing, optimization, and therefore low query response times. On the other hand, there is no guarantee that the data loaded into the data warehouse is up-to-date and we have to update it from time to time. From the perspective of a single user or application, the data warehouse contains a lot of data that is not queried and unnecessarily increases storage space consumption. So, the data warehouse solution is only suitable if we have a sufficiently high number of queries and diverse applications.

2.3 Federated systems

As mentioned above, database literature proposes two main classes of data integration approaches (Figure 6): materialized and virtual data integration. In contrast to materialized data integration approaches, *virtual data integration systems* do not work with copies of the original data but only virtually integrate

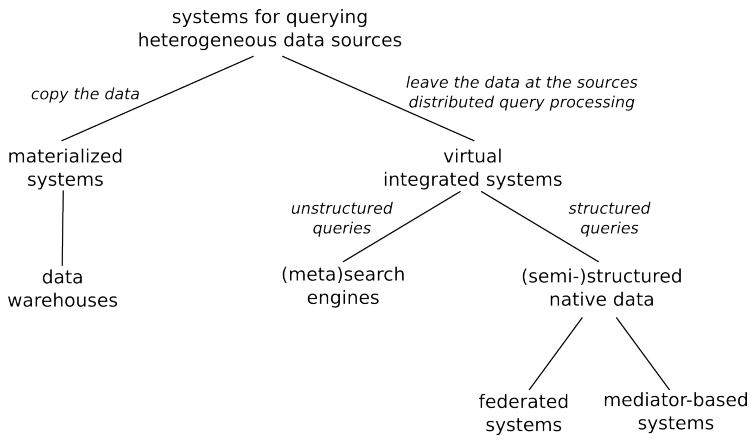

Fig. 6. Classification of data integration approaches

the data. If we are only interested in answering unstructured queries based on keywords, the search engine approach, as a variant of virtual integration systems, is an appropriate solution. We have already discussed several examples of Semantic Web search engines in Section 2.1. Some of them are pure search engines that only work with indexes and output pointers to relevant data at the sources, whereas other Semantic Web search engines rather correspond to data warehouses.

When we are interested in answering structured queries on the most recent version of the data, we need to consider approaches that integrate sources in a way that preserves the sources' autonomy but at the same time allow for evaluating queries based on the data of multiple sources: we distinguish between *mediator-based systems* and *federated systems*.

Mediator-based systems. Classic mediator-based systems provide a service that integrates data from a selection of independent sources. In such a scenario, sources are often unaware that they are participating in an integration system. A mediator provides a common interface to the user that is used to formulate queries. Based on the global catalog (statistics about the data available at the sources), the mediator takes care of rewriting and optimizing the query, i.e., the mediator determines which sources are relevant with respect to the given query and creates subqueries for the relevant sources. In case the sources manage their local data using data models or query languages different from what the mediator uses, so-called wrappers rewrite the subqueries in a way that makes them "understandable" for the sources, e.g., SQL to XQuery. Wrappers also take care of transforming the data that the sources produce as answers to the query into the mediator's model. In practice, the task of the mediator/wrapper architecture is to overcome heterogeneity on several levels: data model, query language, etc.

Federated systems. The second variant of virtual integration systems from a database point of view is *federated systems*. A federation is a consolidation of multiple sources providing a common interface and therefore very similar to the mediator-based approach. The main difference to mediator-based systems is that sources are aware that they are part of the federation because they actively have to support the data model and the query language that the federation agreed upon. In addition, sources might of course support other query languages.

However, from a user's point-of-view there is no difference between both architectures as both provide transparent access to the data. Likewise, in general the Semantic Web community does not distinguish between these two architectures either so that both approaches are referred to as federated systems, virtual integration, and/or federated query processing [51, 54, 63]. Thus, in the following we adopt the notion of a federation from the Semantic Web community and do not distinguish between federated and mediated architectures nor between federators and mediators.

There are some peculiarities unique to the web of linked data that we need to deal with in a distributed setup. Some sources, for instance, do not provide a query interface that a federated architecture could be built upon, i.e., some sources only provide information to dereferenceable HTTP URIs, some sources are only available as dumps, and some sources provide access via SPARQL endpoints.

For the first two cases, we can crawl the data and load it into a central repository (data warehouse) or multiple repositories that can be combined in a federation – comparable to using wrappers for different data sources. If the data is already available via SPARQL endpoints, we only need to register them in the federation and their data can be accessed by the federation to answer future queries. In addition, if we adopt SPARQL as the common protocol and global query language, then no wrappers are necessary for SPARQL endpoints because they already support the global query language and data format.

Another interesting difference between classic federated database systems and federated RDF systems is that for systems using the relational data model, answering a query involves only a few joins between different datasets defined on attributes of the relations. For RDF data, this is more complicated because the triple format requires much more (self) joins. Moreover, some aspects of RDF, e.g., explicit links (`owl:sameAs`), are also aspects that need to be considered during query optimization and execution.

Query processing in distributed database systems roughly adheres to the steps highlighted in Figure 7: query parsing, query transformation, data localization, global query optimization, local query optimization, local query execution, and post-processing. In the following we discuss each of these steps in detail; we first describe each step with respect to classic distributed databases and then compare it to distributed RDF systems.

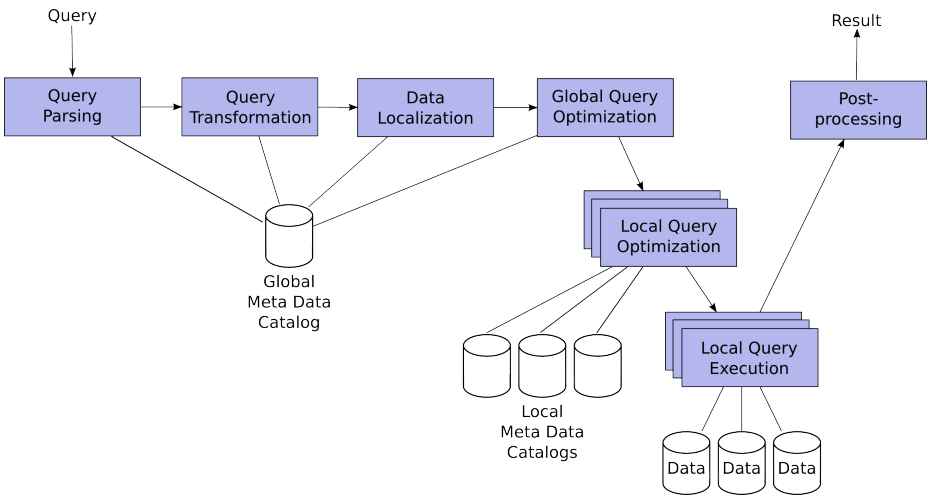

Fig. 7. Steps of distributed query processing

Query parsing. After the query has been formulated by the user, it has to be parsed, i.e., the query formulated in a declarative query language is represented using an internal format that facilitates optimization. For example, a query formulated in SQL is transformed into an algebra operator tree.

Likewise, SPARQL queries can be parsed into directed query graphs: SQGM (SPARQL Query Graph Model) [62]. In any case, after parsing we have a graph consisting of operators that need to be evaluated in order to answer the query – a connection between operators indicates the data flow, i.e., the exchange of intermediate results, between them.

Query transformation. In this step, simple and straightforward optimizations are applied to the initial query that do not require any sophisticated considerations but mostly rely on heuristics and straightforward checks. One aspect is to perform a semantic analysis, e.g., to check whether relations and attributes referred to in the query do actually exist in the schema. The schema information necessary for this step are stored in the global catalog. In addition, query predicates are transformed into a canonical format (normalization) to facilitate further optimization, e.g., to identify contradicting query predicates that would result in empty result sets. Moreover, simple algebraic rewriting is applied using heuristics to eliminate bad query plans, e.g., replacing crossproducts followed by selections with join operators, redundant predicates are removed, expressions are simplified, unnesting of subqueries and views, etc.

Similar transformations can be done for SPARQL queries. For example, we can check for contradictions of conditions in the query that would lead to an empty result set. We can also apply some basic optimizations such as removing redundant predicates and simplifying expressions. In addition, we can also check if the predicates referenced in the query do actually exist, i.e., if there

is any source providing corresponding triples. However, this is only useful for distributed RDF systems that do not traverse links to discover new sources during query processing. As discussed below, the information necessary to perform these checks are part of the global catalog.

Data localization. In the classic data localization step, the optimizer replaces references to global relations with references to the sources' fragment relations that contribute to the global relation. The optimizer simply needs to access the global catalog and make use of reconstruction expressions: algebraic expressions that, when executed, reconstruct the global relations by combining the sources' fragments in an appropriate manner. In consideration of predicates and joins contained in the query, a subset of fragments and therefore sources can be eliminated from the query because the data they provide cannot contribute to the query result.

In case of Linked Data and systems for distributed RDF processing, this step entails the identification of sources relevant to a given query. Whereas we have reconstruction expressions for relational databases, it is more complicated for RDF data because there are no strict rules and or restrictions on which source uses which vocabulary. Thus, what an appropriate optimizer for distributed RDF systems needs to do in this step is to go through the list of basic graph patterns contained in the query and identify relevant sources. The information necessary to perform this task are contained in the global catalg, which we will discuss in detail below.

Global query optimization. In principle, the goal of this step is to find an efficient query execution plan. In classic distributed systems, the optimizer can optimize for total query execution time or for response time. The former is a measure for the total research consumption in the network, i.e., it sums up the duration of all operations necessary to answer the query over all nodes altogether. Response time takes parallel execution into account and measures the time until the results are presented to the user. Optimizing response time can exploit all different flavors of parallelization only when the mediator has some "control" over the sources, i.e., considering direct interaction, communication, and data exchange between any two nodes for optimization.

In federated RDF systems, however, sources are still more autonomous so that some of the options available to classic optimizers cannot be considered, e.g., we cannot decide on a query plan that requires one source to send an intermediate result directly to another source that uses the received data to compute a local join. Thus, techniques commonly referred to as data, query, or hybrid shipping [76] are not applicable in the RDF context. Likewise, the option of pipelining[10] results between operators executed at different sources is hardly applicable.

[10] Result items (tuples) are propagated through a hierarchy of operators (e.g., joins) so that when a tuple is output by an operator on a lower level, it *immediately* serves as input to the upper operator – before all input tuples have been processed by the lower operator.

In general, however, it is efficient in terms of load at the mediator and communication costs when joins can directly be executed at the sources. The problem is that with the restrictions mentioned above joins in distributed RDF systems can only be executed in a completely decentralized fashion at a source s_1 if there is no additional data at any other source s_2 that could produce additional results when joined with part of the data provided by s_1. Thus, even though s_2 can process the join locally it has to return, in addition to the join result, all data that could be joined with other sources' data. Hence, in most cases joins have to be computed in a centralized fashion by the mediator [51, 139]. In that case, the optimizer applies techniques for local query optimization as discussed in Section 1.

Another option to process joins efficiently is to use a semijoin [76] algorithm to compute a join between an intermediate result at the mediator and a data source [51]. The mediator needs to compute a projection on the join variables and extract all variable bindings present in the intermediate result. For all these variable bindings, the mediator needs to query the remote data source with respect to the join definition and retrieve the result, which consists of all triples matching the join condition. So, the mediator retrieves all relevant data to perform the join locally. Unfortunately, SPARQL does not support the inclusion of variable bindings in a query so that the only alternative is to include variable bindings as filter expressions in a query, which for a large number of bindings might blow up the size of the query message. The alternative of sending separate messages for each binding [115] results in a high number of messages and therefore increases network load and query execution costs.

A heuristic to minimize query execution costs is to minimize the size of intermediate results. Hence, optimizers for federated RDF systems also try to find a plan that minimizes the size of intermediate results. Theresfore, the optimizer considers additional statistics about selectivities and cardinalities provided by the global catalog (see below).

Another standard technique for every optimizers is to apply heuristics to reorder query operators in a way that intermediate results are small, i.e., pushing highly selective operators downwards in the query graph so that they are executed first. Considering RDF data, this means to push down value constraints – ideally into subqueries executed at the sources to reduce the amount of exchanged data.

Another important and related issue for standard optimizers is join order optimization. The goal is to find the order of joins that minimizes execution costs. This process heavily relies on statistics (global catalog) to estimate the size of intermediate results. Obviously, it is beneficial to execute joins first that produce small outputs so that all other joins can be computed more efficiently. For optimization in distributed RDF systems, join order optimization corresponds to determining the order of evaluating basic graph patterns. To estimate the size of intermediate results, the optimizer needs detailed statistics that allow for cardinality estimation. Histograms are very common for relational distributed systems and can also be useful in the context of federated RDF systems. A

simple heuristic that can be applied without detailed statistics is variable counting [136], which estimates the selectivity of basic graph patterns in dependence on the type and number of unbound components.

In consideration of all these different possibilities for query optimization, the optimizer has to explore and evaluate a potentially high number of alternative query plans that all compute the same result. For exploration, a common technique is the application of dynamic programming for plan enumeration, which enables an exhaustive search of all query plans. In order to decide on the benefit of each plan, the optimizer has to determine its costs using an appropriate cost model [51, 105, 139]. For this purpose, the optimizer again needs statistics about the data, selectivity, cardinality estimations about intermediate results. In addition, it is worthwhile to consider network latencies as well.

The current standard for distributed RDF optimizers is to optimize a query and to determine a query plan for each query individually at runtime. This can be improved by adopting more techniques from classic distributed database systems that pre-optimize (partial) execution plans for frequently issued queries or subqueries contained in many queries, e.g., two-step-plans [76].

As a high number of sources provide potentially relevant data for a given query, approximation by reducing the number of queried sources is an appropriate technique to reduce the overall execution costs. This can be achieved by ranking sources based on triple and join pattern cardinality estimations and prune all but the top-ranked sources from consideration [59].

Global catalog and indexes. As we have seen above, the global catalog plays an important role for query optimization. It might contain information about specific sources such as vocabulary, supported predicates, network latencies, etc. In addition, for the distributed setup and especially for cardinality estimation, the catalog has to contain statistics about the data, e.g., in the form of indexes. In constrast to indexes that are applied in the context of centralized systems (complete indexes), these indexes cannot provide the same level of details because of the amount of data from all sources altogether. Therefore indexes and statistics about the data suitable for distributed optimization have to abstract the data in a way that allows for index size compression. The goal is to find a suitable trade-off between the level of detail and memory space consumption – the general rule is the more detailed an index is, the more accurate are cardinality estimations but the higher is the memory consumption.

A widely used approach for indexing is *schema-level indexing*, i.e., predicate URIs and the types of instances. Whereas types and predicates that occur only rarely represent good discriminators to detect relevant sources for a given query, frequently used types and predicates that almost every source provides are only of little use for optimization, e.g., `rdfs:label`. The disadvantage is that this index cannot be used for basic graph patterns with variables in the predicate position. An alternative to indexing predicates and types detached from the structure of the overall RDF graph is indexing paths of predicates [139]. Moreover, when considering the RDF data of a source as a graph, it might also be useful to index frequent subgraphs [51, 89, 146].

Another kind of indexes are *inverted URI indexes*. They index the data on instance level by indexing all URIs occuring in a data source. This kind of index allows the query processor to identify all sources which contain a given URI and thus potentially contribute bindings for a triple pattern containing that URI.

There are also indexes indexing data on both instance-level and schema-level [59]. They make use of data structures known from classic relational database: histograms. As these data structures have originally been developed to summarize informtion about numerical data, hash functions are applied to all elements of a triple so that triples are transformed into numerical space. One-dimensional histograms index each component (subject, predicate, object) in separate. But when considering each of these three dimensions in separate, the optimizer looses a great potential for optimization because the combination of instances is much more selective and therefore more useful for query optimization. Thus, multidimensional histograms are applied, i.e., three-dimensional histograms. An alternative to histograms, other structures that efficiently summarize multidimensional data can be applied, e.g., QTrees [59].

Local query optimization and execution. When the global optimizer has decided for a specific query execution plan, subqueries are extracted and sent to the sources for local execution. They apply the same optimization techniques as for local queries, so that the received query is treated like a query issued directly at the source. Consequently, it is optimized and executed using the techniques discussed in Section 1.

Post-processing In this last step the partial results received from the sources are combined into the final result. For simple distributed queries, the post-processing might simply consist of a union operation. For more complex queries, however, post-processing is much more complex and costly because all operations that could not be executed at the sources have to be processed by the mediator after retrieving the data from the sources. As discussed above, this is particularly true for joins for which multiple sources provide relevant data. In addition, it might be necessary to remove duplicates in the result set, which represents the very last operation for post-processing.

To conclude the section about federated systems and federated RDF processing, we discuss some systems that have been proposed in this context.

SemWIQ. The Semantic Web Integrator and Query Engine [79], SemWIQ in short, uses an architecture based on the mediator/wrapper approach, i.e., wrappers are used to enable the participation of sources using other data models, e.g. relational databases. All registered data sources must either be connected by a SPARQL-capable wrapper or support the SPARQL protocol directly.

Queries are formulated in SPARQL, the parser computes a canonical query plan, which is optimized by the federator/mediator. The optimization process only considers very basic statistics, such as a list of classes and the number of

instances a data source provides for each class as well as a list of predicates and their occurrences. For query optimization, the federator analyzes the query and scans the catalog for relevant registered data sources. The resulting plan is executed by sending subqueries to the sources (via wrappers or SPARQL endpoints).

The system requires that every data item has an asserted type, i.e., for a query it requires type information for each subject variable in order to be able to retrieve instances. As a consequence, there are some restrictions with respect to query formulation, e.g., all subjects must be variables and for each subject variable its type must be explicitly or implicitly defined. The optimizer then uses this information to look for registered data sources providing instances of the queried types. More sophisticated optimization techniques, such as the push-down of joins, have been proposed as future work.

DARQ. DARQ [115] (Distributed ARQ) is a federated system of SPARQL endpoints that allows distributed query processing on the set of registered endpoints. It also adopts a mediator-based approach and assumes that sources that do not support SPARQL themselves are connected to the federation using wrappers.

Data sources are described using service descriptions (represented in RDF). These descriptions contain capabilities, i.e., constraints expressed with regular SPARQL filter expressions. These constraints can express, for instance, that a data source only stores data about specific types of resources. For sources with limitations of access patterns, e.g., allowing lookups on personal data only when the user can specify the name of the person of interest. DARQ supports this by defining patterns in the service descriptions that must be included in the query. To provide the query optimizer with statistics, service descriptions contain the total number provided by a data source and optionally information for each capability, e.g., the number of triples with a specific predicate and the selectivities (bound subject/object) of a triple pattern with a specific predicate.

Queries are formulated in SPARQL, parsed, and handed to the query planning and optimization components. DARQ uses a cost-based query optimization technique relying on the statistical information provided as service descriptions and capabilities.

A SPARQL query contains one or more filtered basic graph patterns (triple patterns). DARQ performs query planning for each basic graph pattern in separate. By comparing the triple patterns of the query against the capabilities of the service descriptions, the system can detect a set of relevant sources for each pattern. As this matching procedure is based on predicates, DARQ only supports queries with bound predicates.

After having determined the relevant sources, subqueries are created, one for each basic graph pattern and data source matching. Thus, the system might create multiple subqueries that are to be sent to the same data source. In that case, the subqueries can be combined into one message.

Based on these subqueries the query optimizer considers limitations on access patterns and tries to find a feasible and efficient query execution plan. For logical

query optimization, heuristics are used that try to simplify the query and reduce intermediate result sizes, e.g., push value constraints into subqueries, which corresponds to pushing down selections in classic query optimization. In order to decide for a specific implementation to process joins (nested loop join or bind join), the optimizer estimates the result size of joins based on the statistics given in the service descriptions. The optimizer chooses the implementation with the least estimated transfer costs (computed based on the result estimates).

In the end, subqueries are executed at the sources and remaining operations are executed by the federator.

Hermes. Hermes [147] is a system based on a federated architecture that has a slightly different focus than other systems discussed so far. Queries are not formulated using SPARQL. Instead, the user enters a keyword query and Hermes tries to translate the keyword query into a structured query (SPARQL). The query is decomposed into subqueries, which are executed at the sources.

In order to achieve this, a number of indexes are created: a keyword index (indexing terms extracted from the labels of data graph elements), a structure index (information about schema graphs, i.e., relations between classes derived from the connections given in the data graph), and a mapping index (information about mappings on data- and schema-level, pairwise mappings between elements).

After having received the input keywords, relevant sources are determined using the keyword index. The retrieved keyword elements are combined with schema graphs received from the structure index to find substructures that connect all keyword elements. A ranking function is used to rank the computed query graphs so that the user can choose some of them for execution.

The selected query is decomposed into subqueries, each of which is answered by a different data source. Optimization uses the same techniques as DARQ [115]. Before sending the subqueries to the sources, they are mapped to the query format supported by the receiving data source. After execution, the results received from the sources are combined.

Other systems. Virtuoso [41], a native quad store, provides the option to consider remote sources for query execution. The system dereferences URIs and holds the retrieved data in a cache for future queries.

Dartgrid [25] is a system for SPARQL queries over multiple relational databases. One of the main components is the semantic registration service which maintains mappings from the schemas of registered data sources to the internal ontology. An query interface based on forms and ontologies helps users to construct semantic queries formulated in SPARQL. Queries are translated using the mapping information into SQL queries that can be executed on the sources.

The networked graphs approach [126] allows users to reuse and integrate RDF content from other sources. It allows users to define RDF graphs by extensionally listing content and by using views on other graphs. These views can be used to include parts of other graphs. Networked graphs are designed for distributed

settings and are exchangeable between sources. However, the paper focuses more on semantics and reasoning than on aspects of query execution.

[139] presents an approach for querying distributed RDF data sources and introduces index structures, a cost model, and an algorithm for query answering. The approach supports distributed SeRQL path queries over multiple Sesame RDF repositories using a special index structure to determine the relevant sources for a query.

2.4 Discovering new sources during query processing

In contrast to the classic understanding of federated databases, processing Linked Data arises additional challenges that we have not discussed in detail above, e.g., by dereferencing URIs and considering the returned document as a new virtual "data source" not all sources are known in advance and available for indexing. Furthermore, approaches that index a static set of sources, i.e., all approaches we have discussed so far, have to recreate their indexes from time to time in order to reflect recent updates to the sources, which means that some of the information provided by the index might be out-of-date.

Pure. Some strategies for query processing over Linked Data rely on the principle of following links between sources [61]. An advantage in comparison to federated architectures is that sources that are not accessible via SPARQL endpoints can be considered. Another advantage is that users can retain complete control over the data they provide.

The query is executed without a previous query planning or optimization step. First, the system retrieves data from the sources mentioned in the query. The data is partially evaluated on the retrieved data so that relevant source URIs and links can be identified. The system uses these URIs to retrieve more data. It iteratively evaluates and discovers further data until all sources found to be relevant have been processed.

The peculiarity of this approach is the intertwining of the two phases: query evaluation and link traversal. Previous work [17,94] kept these two phases in separate by first retrieving the data and then evaluating the query on the retrieved data.

Hybrid. It is also possible to combine federated query processing with active discovery of unknown but potentially relevant sources [78]. The main assumption is that knowledge about some sources is available for query planning and optimization. During query execution, sources are retrieved, new sources are iteratively discovered, the query plan is reoptimized and partially executed until all relevant sources have been processed.

2.5 Systems based on the P2P paradigm

As we have seen, distributed processing of RDF data can be realized by adopting the federated database system architecture and developing efficient solutions for

problems that come along with the characteristics of Linked Data. But there is another important class of distributed systems that Linked Data processing can also benefit from: peer-to-peer (P2P) systems.

P2P systems are networks of autonomous peers that are connected through logical links, which express that a pair of peers "know" each other, i.e., they can exchange messages and data and are referred to as *neighbors*. A pair of peers without a link do not know each other and can therefore only contact each other if they find a path of links via intermediate peers that connects them.

In general, sources/peers in a P2P network have a higher degree of autonomy in comparison to sources in distributed database systems. In pure P2P networks there is no central component, i.e., no federator or mediator, that could be used for query planning and optimization. Instead, the behavior of the whole system is the consequence of local interactions between peers.

Whereas sources in the context of distributed databases are considered to be rather stable and available, P2P systems assume a higher degree of dynamic behavior, i.e., they assume that peers might join and leave the network at any time. Even though a peer leaves the network, the system should still be able to answer queries – even though the data provided by the peer that left the network is (momentarily) unavailable. Moreover, each peer in the network might issue queries and participate in answering queries. There are different classes of P2P systems: *centralized P2P*, *pure P2P*, *hybrid P2P*, *structured P2P*.

Centralized P2P systems. For centralized P2P systems, like Napster[11], there is a central component providing a centralized index that is used to locate relevant data to a given query. So, a user query issued at a peer is sent to the central component, which uses the index to find peers providing the queried data. This information is sent back to the querying peer, which then directly communicates with other peers to access the queried data.

Pure/unstructured P2P systems. As the centralized server represents a bottleneck and a single point of failure, pure P2P system strictly avoid peers with special roles. Instead, all peers are considered equal and query processing is realized by *flooding*, i.e., a peer with a local user query forwards the query to all its neighbors, they proceed the same so that the query finally reaches all the peers in the network. The answers to the query are routed back to the query initiator, which can then directly communicate with peers providing relevant data. In analogy to structured P2P systems, which we will discuss below, pure P2P systems are often referred to as *unstructured P2P systems*.

Hybrid P2P systems. The problem with flooding is that query processing consumes much bandwidth and weak/slow peers represent the bottleneck. So, hybrid P2P systems use super-peers to counteract this problem. Super-peers are strong peers that form an unstructured P2P network. Weaker peers are connected to a super-peer as leaf nodes and form a centralized P2P system.

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

Structured P2P systems. All the P2P system discussed so far assume that the data shared in the network remains at the peers that "own" them. However, in structured P2P systems, a global rule is used to redistribute the data among peers in the system. In most cases a hash function is used for this purpose so that each peer is responsible for a specific hash range and therefore for all the data with hash values in that range. Peers are arranged in a logical overlay structure, e.g., a logical ring [137], that is used to organize the peer in way that alleviates efficient lookup. For query processing peers use the globally known rule according to which the data has been distributed in the first place, e.g., peers compute hash values for the queried data and use the overlay network to locate peers responsible for overlapping hash ranges.

After having introduced the basic types of P2P systems, let us now discuss some approaches that apply these concepts to the Semantic Web and RDF data.

Edutella. An early approach that combined the two paradigms RDF and P2P is Edutella [98]. Edutella assumes that all resources maintained in the network can be described with metadata in RDF format. All functionality in the Edutella network is mediated through RDF statements and queries on them.

Peers participating in an Edutella network might use different schemas so that Gnutella applies the mediator-wrapper approach based on a common data model and a common query exchange format to overcome heterogeneity. A peer that wants to participate in the network, registers at a so-called mediator peer. Peers register the metadata schemas they support and in this way indicate which queries they can answer. Queries are sent through the Edutella network to the subset of peers that have registered with the corresponding schema. The resulting RDF statements are sent back to the requesting peer. To broaden the search space, mediators provide a service that translates queries over one schema into queries over another schema.

Because of the mediators that mediate between clusters of peers supporting different schemas, the overall architecture can be considered a *hybrid P2P system.*

GridVine. GridVine [5] uses P-Grid [4] to organize peers in a *structured P2P* overlay network, which is used for communication and interaction between peers. Data is indexed and stored in the standard way of structured P2P systems, i.e., each peer maintains a local database at the semantic layer to store the triples whose keys are contained in the key range the peer is responsible for. GridVine also supports sharing schemas by associating schemas with unique keys and storing them in the overlay network.

GridVine supports queries based on basic graph patterns and exploits pairwise mappings (OWL statements relating similar classes and properties) between different schemas to overcome schema heterogeneity and evaluate queries against schemas they were originally not formulated against. Mappings are also stored in the network.

In order to locate peers providing relevant data for a given query, or a basic graph pattern respectively, each triple is indexed and stored three times – once for each component of the triple.

RDFPeers. RDFPeers [20] is similar to GridVine but uses a different semantic overlay network for storing, indexing, and querying RDF data. To efficiently answer multi-attribute and range queries, RDFPeers relies on a multi-attribute addressable network (MAAN) [21], which extends Chord [137] – *structured P2P*.

Each triple is stored three times applying hash functions to subject, predicate, and object. The system's query processing capabilities are very similar to the ones of GridVine. It supports triple pattern queries, disjunctive and range queries, and conjunctive multi-predicate queries using RDQL.

QC and SBV. Another approach for evaluating conjunctive queries of triple patterns over RDF data using *structured* overlay networks is proposed in [83]. It uses Chord [137] and also stores each triple three times (subject, predicate, object).

This approach proposes two algorithms for query processing: *query chain* (QC) and *spread by value* (SBV). The main idea of the query chain algorithm is that intermediate results flow through the nodes of the chain. First, responsible nodes are determined for each triple pattern contained in the query – exploiting constants contained in the patterns and using the overlay structure to identify relevant peers. The found responsible peers form a chain and exchange messages to answer the query.

The spread by value algorithm constructs multiple chains for each query that can be processed in parallel. Query processing starts at a node responsible for the first query pattern, which uses values of matching triples to forward the query to nodes providing data for these values.

Other approaches. YARS2 [60] uses the P2P paradigm (*structured P2P*) in a different way than the approaches discussed so far. As mentioned above in Section 2.1, it comes in combination with the search engine SWSE. YARS2 does not distribute the data in the structured overlay network but an index structure. More specifically, YARS2 uses indexes in six different orderings of triple components and distributes this index according to a hash function.

KAONp2p [55] also suggests a P2P-like architecture for query answering over distributed ontologies. Queries are evaluated against resources, which are integrated using a virtual ontology that logically imports all relevant ontologies distributed over the network.

3 Scalable Reasoning with Uncertain RDF Data

As we have seen in the previous chapter, state-of-the-art SPARQL engines for RDF data (see, e.g., [3, 99]) primarily focus on conjunctive queries on top of a

relational encoding of RDF [1] data. They often employ a so-called "triple-store" technique by indexing or slicing the data according to various permutations of the basic subject-predicate-object (SPO) pattern. These engines generally follow a deterministic data and query processing model and do not have a notion of uncertain reasoning or probabilistic inference.

In this chapter, we aim to devise possible directions for scalable reasoning with uncertain RDF knowledge bases, following ideas from probabilistic databases, logic programming, and recent imperative programming platforms for inference in probabilistic graphical models. We focus on RDF as our basic data model and SPARQL as the default query language for RDF. Consequently, the forms of uncertain reasoning we consider here focus on SQL-style queries in probabilistic databases, Datalog-style reasoning using Horn clauses as rules, as well as some probabilistic extensions to logic programming.

By default, an RDF graph itself is schemaless. RDFS [1] thus introduces basic facilities for imposing schema information in terms of a class hierarchy. Intuitively, entities that belong to the same instance of a class, i.e., entities of the same RDF *type*, share common properties. In RDFS, instances of classes, class memberships and class properties are transitively inherited to resources in subclasses via the *type*, *subClassOf* and *subPropertyOf* relations, respectively. More generally, rule-based reasoning using Datalog-style Horn clauses subsumes *type*, *subclassOf* and *subPropertyOf* inferences in RDF/S and also captures some of the constraints expressible in OWL [7] (e.g., the *transitive* or *functional* properties of predicates). Transitivity, for example, can very easily be expressed via first-order predicate logic (specifically with rules in the form of Horn clauses), which require a from of logical reasoning in order to check consistency or to compute entailment. Considering relations as logical predicates and facts as literals, we thus also briefly investigate the relationship to logic programming and its probabilistic extensions in this chapter.

As a (more popular) counterpart to probabilistic logic programming, probabilistic databases [32] investigate query processing techniques for structured (SQL) queries over relational data with fixed schemata. While the general idea of including probabilistic models into databases is not new and leads back to more than 30 years of research (see, e.g., [24]), in particular recent works in this context have provided us with a rich body of literature and have led to the development of a plethora of systems, which all aim at the scalable management of uncertain relational data. Ultimately, these approaches however need to face the very same scalability and complexity issues as any approach dealing with inference in probabilistic graphical models.

As this part of the lecture focuses on database-style infrastructures for the management of uncertain RDF data, we do not go into details on probabilistic extensions to OWL and its variants based on description logics (OWL-DL). For probabilistic description logics [87] (PDL), we refer the reader to [138], which provides a very good overview of PDL and related approaches, which has found wide-spread acceptance in the semantic web community. PDL generalizes the description logic $\mathcal{SHOIN}^{(D)}$ and can thus express, for example, functional rules.

Reasoners such as PRONTO[12] can be used to decide consistency or to compute entailment with probabilistic bounds.

3.1 Probabilistic Databases

Approaches for managing uncertainty in the context of probabilistic databases [8, 13,30,31,132] focus on relational data with fixed schemata, and they often employ strong independence assumptions among data objects (the "base tuples"). SQL is used as the default query language for running queries, defining views, or triggering updates to the data.

Most probabilistic databases adopt the *possible worlds* [6] model as basis for their data model and for defining the semantics of queries. Intuitively, every tuple in a probabilistic database corresponds to a binary random variable which may exist in the database with some probability. A probabilistic database thus encodes a large number of possible instances of deterministic databases, where each deterministic instance contains a different combination (i.e., possible world) of tuples. Each such possible world has a probability between 0 and 1. The probabilities of all worlds form a distribution and sum up to 1. The *marginal probability* of a tuple can be obtained by summing up the probabilities of all worlds which contain that tuple, which leaves confidence computations in probabilistic databases #P-complete [30, 120] in the general case. The semantics of queries is then formally defined by running the query against each of these instances individually, and by encoding the results obtained from each of the individual instances back into the probabilistic database. While often the actual data computation can be carried out directly along with the SQL operators, different inference techniques for confidence computations may have to be implemented as separate function calls (e.g., as stored procedures [96]).

In addition to this basic uncertainty model, the ULDB [13] data model (for "Uncertainty and Lineage Database") provides a lineage-based representation formalism for probabilistic data, which has been shown to be *closed and complete* for any combination of SQL-style relational operators and across arbitrary levels of materialized views. Here, the lineage (aka. "history" in [132] or "repair-key" operator in [8]) of a derived tuple (or an entire view) is captured as a Boolean formula which recursively unrolls the logical dependencies from the derived tuple back to the base tuples. The lineage of a derived tuple may never be cyclic, but it may impose a DAG structure over the derivation of a tuple. Thus, being a form of a directed (but acyclic) graphical model, probabilistic inference in ULDBs remains #P-complete for general SQL queries.

On the other hand, considering restricted classes of SQL queries and corresponding query plans, where exact confidence computations remain tractable, has led to a notion of "safe plans" in [30]. Intuitively, an entire query plan is safe, if all query operators take only independent subgoals as their input, which guarantees a hierarchical derivation structure (i.e., tree-shaped lineage) of all tuples involved in a query result. Following this idea on a more fine-grained

[12] `http://pellet.owldl.com/pronto/`

(i.e., on a per-tuple rather than a per-plan) level, [129] considers a class of so-called "read-once" functions, where the Boolean lineage formula can be factorized (in polynomial time) into a form, where every variable appears at most once. Moreover, efficient top-k query processing [116, 134] and unified ranking approaches [81] have investigated different semantics of ranking queries over uncertain data. Recently, the modeling of correlated tuples [127] with the explicit usage of probabilistic graphical models such as Bayesian Nets [18,151] and Markov Random Fields [128] has found increasing attention also in the database community. In [70], the authors define a class of Markov networks, where query evaluation with soft constraints can be done in polynomial time, while the case with hard constraints is considered separately in [31]. Also in the context of these graphical models, lineage remains the key for a closed representation formalism [71].

While most probabilistic database systems provide extensions to the DDL (data definition language) part of the SQL standard to define dependencies at the schema level, only fairly few works explicitly tackle the DML (data manipulation language) part of SQL, including data modifications such as updates or deletes [66, 125]. Most probabilistic database approaches focus on SELECT-PROJECT-JOIN (SPJ) query patterns, where query processing bears a number of interesting analogies to inference in probabilistic graphical models.

MystiQ. The MystiQ system [16,30,117] developed at the University of Washington provides support for a wide range of SQL queries over uncertain and inconsistent data sources. On the DML part, MystiQ introduces the notion of an *approximate match* operator between a query attribute and a data attribute, which can be evaluated by data-type-specific built-in functions. On the DDL part, MystiQ supports so-called *predicate functions*, which define how probabilities should be generated for an approximate match on such an attribute. Moreover, in the spirit of functional dependencies in deterministic databases, MystiQ allows for the specification of global constraints among attributes at the schema level. Unlike classic functional dependencies, these constraints can be specified to be either strict (e.g., a person may have only exactly one date-of-birth) or soft (e.g., most people have different names). A violation of a constraint mutually affects the confidences of all tuples involved in the conflict. In further works, the authors present an efficient top-k algorithm (coined "multi-simulations") for a class of SELECT DISTINCT queries which exhibit a DNF structure as lineage. Multi-simulation works by running multiple Monte Carlo simulations [73] in parallel until the lower confidence bounds of the top-k answers to the query have sufficiently converged, in order to distinguish them from the upper confidence bounds of the non-top-k answers.

Trio. Based on the ULDB data model, the Trio [96] system developed at Stanford University provides an integrated approach for the management of *data*, *uncertainty*, and *lineage*. Trio is implemented on top of a conventional database system (PostgreSQL) and employs an SQL-based rewriting layer for the Trio

query language (TriQL) into a series of relational queries and calls to stored procedures. As core of its data model, Trio adopts the notion of X-tuples [123] for mutually exclusive tuple alternatives, Boolean lineage formulas, *maybe* annotations (which indicate the possible absence of the tuple in the uncertain database), and confidence values that may be attached to each tuple alternative. In Trio (and ULDBs), lineage enables for the complete decoupling of data and confidence computations, which may yield significant efficiency benefits for query processing. Later extensions to Trio have investigated in more detail how to exploit lineage for probabilistic confidence computations [124] and data updates [125].

MayBMS. The MayBMS [8,66] system initially developed at Saarland University and then at the Cornell database group is designed as a completely native extension to PostgreSQL. Based on the encoding scheme of conditional tables (C-tables) [67,123], the current MayBMS system employs a compact form of schema decompositions (coined "U-relations"), which results in a succinct encoding of an uncertain relation with independent attributes. Another focal point of the MayBMS project includes the investigation of foundations for query languages in probabilistic databases in analogy to relational algebra and SQL [75]. Ongoing research issues in MayBMS include query optimization, an update language, concurrency control and recovery, and the design of generic APIs for uncertain data. MayBMS is the basis for the SPROUT system discussed in Section 4.

Orion. Inspired by large-scale sensor nets, the Orion [131,132] system developed at Purdue University also investigates continuous probability density functions (PDFs) in combination with SQL-style relational query processing techniques. Originally inspired by the application of sensor networks, the current Orion 2.0 prototype has built-in support for a number of continuous (e.g., Gaussian, Uniform) and discrete (e.g., Binomial, Poisson) distributions, which are treated symbolically at query processing time. If the resulting distribution of an SQL-style operation cannot be represented by a standard distribution anymore, Orion switches to an approximate distribution using histograms and sampling. Further features of Orion include the handling of correlations among attributes, which are captured as explicit joint distributions among the correlated attributes, and the handling of missing (i.e., incomplete) data, which is implemented by allowing for partial PDFs whose confidence distributions may sum up to less than 1. Using a form of query history, the Orion data model is closed under common relational operations and consistent with the possible-worlds semantics.

PrDB. One of the most active current probabilistic database projects is the PrDB [128] system developed at the University of Maryland. Unlike other probabilistic database approaches, PrDB employs undirected probabilistic graphical models, specifically Markov Random Fields, as basis for handling correlated base tuples. The graphical model is stored directly in the underlying database system

in the form of *factor tables*, which capture correlations among tuples and serve as input for the probabilistic inference algorithms. Due to the generality of this data model, PrDB incorporates a large variety and optimizations for both exact and approximate inference, including variable elimination [36], the reuse of shared factors, and Gibbs sampling [47] in the general case. In addition, bisimulation [72] is shown to significantly speed up inference for DAG-shaped queries in this context. Also here, lineage, in the form of Boolean formulas that capture the logical dependencies of derived data objects (tuples) back to the base tuples, is the key for a closed and complete representation model. Moreover, the efficient processing of lineage for probabilistic inference under this data model has been studied in [71].

3.2 Logic Programming & Rule-based Reasoning

The semantic web has led to the development of a plethora of rule-based and description-logic-based reasoning engines, including reasoners like Sesame, Jena, IRIS, Bossam, Prova, and many more[13]. Besides classical logic programming frameworks based on Prolog and Datalog, these engines specifically focus on different ontological reasoning concepts based on the RDF/S standards and the DL (based on description logic), RL (supporting first-order Horn rules) and EL (supporting rules with restricted existential quantifiers) fragments of OWL.

Besides the different grounding techniques and varying expressiveness of the ontological concepts these reasoners support, an important semantic distinction can be made in the way these engines handle negation. *Negation-as-failure* [28] is the most common semantics for handling negation in rule antecedents (bodies), which is also the default semantics used in most Prolog and Datalog engines. Intuitively, a negated literal in the body of the rule is grounded if no proof for the literal can be derived from the knowledge base. As a form of *closed-world assumption* this semantics can lead to non-monotonic inferences when new information (i.e., facts or rules) are entered into the knowledge base, such that the negation of the literal no longer holds. To tackle this issue, the *well-founded negation* and *stable-model semantics* have been proposed to handle negation in rule antecedents. We refer the reader to [45, 102, 143] for details. However, care is advisable when reasoning with recursive rules; already plain Datalog without negation has been shown to be EXPTIME-complete for non-linear, recursive programs (i.e., for predicates with more than one argument and rules with more than one recursive predicate in their antecedents) and still PSPACE-complete for linear, recursive programs, respectively [142].

To evaluate the performance of these engines, a number of benchmarks have been defined, out of which the most prominent ones are probably [145] for RDF/S, as well as the Lehigh University Benchmark (LUBM) [52] and the more recent OpenRuleBench [82] initiative. In the following subsections, we provide a brief overview of the top-performing engines evaluated in OpenRuleBench: OntoBroker, XSB, Yap, and DLV.

[13] See `http://www.w3.org/2007/OWL/wiki/Implementations` for a current overview.

OntoBroker[14] is designed as a Java-based, object-oriented database system. It has been originally developed at the AIFB Karlsruhe and meanwhile turned into a commercial product. It follows a bottom-up, deductive grounding technique and supports Magic-Sets-based rule rewriting [12], as well as a cost-based query optimizer (similar to a relational database system)—a feature that many other rule engines lack. OntoBroker supports the well-founded negation.

XSB[15] is an open-source Prolog engine implemented in native C. In addition to a top-down processing of Prolog or function-free Datalog programs, it can also be used in a deductive (i.e., bottom-up) database fashion, using advanced grounding techniques based on tabling (aka. "memoing") [153]. Through tabling, XSB is able to terminate even for cases when many Prolog engines (based on top-down SLD resolution [77]) run into cycles. Unlike most Prolog engines, XSB supports the well-founded negation and (via a plug-in) also the stable-model semantics.

Yap[16] is a highly optimized Prolog engine developed at the Center for Research in Advanced Computing Systems and the University of Porto. Like XSB it supports advanced grounding techniques based on tabling, but (unlike XSB) it can also create indices for faster data access on-the-fly, when it determines that a particular index may speed up the access to a large amount of data. It however supports only negation-as-failure, but not the well-founded negation nor the stable-model semantics.

DLV[17] is a bottom-up rule system implemented in C++. It is unique in that it allows for a form of disjunctive Datalog programming with disjunctive rule consequents (heads). Moreover, it is the only system along theses lines which supports the stable-model semantics. As additional feature, DLV has built-in support for propositional reasoning over the grounded model using Max-SAT solving and similar techniques. As most of these engines, it is a pure query engine, i.e., it does not support incremental updates to the knowledge base.

3.3 Combining First-order Logic and Probabilistic Inference

In the following, we assume a knowledge base to consist of a finite set of first-order logical formulas and a finite set of (potentially typed) entities. We focus only on reasoning techniques which work by grounding (i.e., by instantiating) the first-order rules, and which again result in a finite set of propositional formulas. This class of rules conforms to a subset of first-order logic that is generally referred to as the Bernays-Schönfinkel-Ramsey class, which is decidable and can be evaluated by grounding the first-order formulas. In some settings, we might want to distinguish between *soft rules*, which may be violated and thus typically have a confidence weight associated with them, and *hard constraints*, which may

[14] `http://www.ontoprise.de/`
[15] `http://xsb.sourceforge.net/`
[16] `http://www.dcc.fc.up.pt/~vsc/Yap/`
[17] `http://www.dbai.tuwien.ac.at/proj/dlv/`

not be violated. A wide variety of grounding techniques exist, each leading to a different reasoning semantics and potentially different answers to queries. What all grounding techniques have in common is that they bind the variables in the rules with the entities contained in the knowledge base in order to obtain a (grounded) set of propositional formulas. In the following, we will consider the actual grounding procedure mostly as a black-box function.

We formally call a knowledge base *inconsistent* if the conjunction of all propositional statements that can be derived from it (e.g., via grounding the rules) evaluates to *false*. Moreover, we call a knowledge base *unsatisfiable* if there exists no truth assignment to variables such that the hard constraints are satisfied.

Propositional Reasoning. Classic (deterministic) approaches to handling inconsistencies in a set of propositional formulas are based on the Boolean satisfiability problem, generally known as SAT. Although the general SAT problem is NP-complete, many real-world problems have actually been shown to be "easy" to solve even for thousands of Boolean variables and many tens of thousands of constraints. Moreover, in recent years the field of SAT solving has made great progress in developing strategies, which allow for tackling also non-trivial instances of the SAT problem very efficiently. Introducing soft rules, on the other hand, leads to a weighted form of the satisfiability problem, generally known as the *maximum satisfiability problem* (Max-SAT). Here the goal is to find a truth assignment to variables which maximizes the aggregated weights (typically the sum) of the formulas which are satisfied by this assignment. Finding the optimum solution in Max-SAT solving however is also NP-complete. More specifically, the maximum satisfiability problem over Horn clauses (coined Max-Horn-SAT) has been studied in detail in [69]. In [48], the authors provide a 3/4 approximation algorithm for the weighted Max-SAT problem over Boolean formulas in conjunctive normal form (CNF). None of these classic (deterministic) Max-SAT solvers however considers a distinction between soft and hard constraints. Thus, recently a family of stochastic Max-SAT solvers has been introduced with WalkSAT [154] and MaxWalkSAT [74], which apply different strategies in order to explore multiple possible worlds to more accurately approximate the optimum solution. A counterpart to Max-SAT solving from a probabilistic perspective is Maximum-a-Posteriori estimation (or "MAP-inference") [133], which selects the most likely mode, i.e., the most likely assignments to variables, according to their posteriori distribution. Recently, for example in the context of information extraction, grounding a set of first-order formulas and post-processing the propositional formulas by a Max-SAT solver has been applied successfully in the SOFIE [141] and PROSPERA [97] projects, in order to automatically populate the YAGO [140] knowledge base[18].

Markov Logic Networks. Statistical relational learning (SRL) [46] has been gaining an increasing momentum in the machine learning, database, and semantic web communities, with Markov Logic Networks [118] probably being the

[18] `http://www.mpi-inf.mpg.de/yago-naga/`

most generic approach for combining first-order logic and probabilistic graphical models into a unified representation and reasoning framework. Intuitively, Markov Logic works by grounding a set of first-order logical rules against a knowledge base, and by sampling states ("worlds") over a Markov network that represents the grounded (i.e., propositional) formulas. Inference in probabilistic graphical models in general is #P-complete. Therefore, Markov Chain Monte Carlo (MCMC) [47, 114, 133] denotes a family of efficient sampling algorithms for probabilistic inference in these graphical models, with Gibbs sampling [47] being one of the most widely used sampling technique, which is also employed in Markov Logic.

Markov Logic however does not easily scale to very large knowledge bases. Grounding a first-order Markov network works by binding all entities (constants) to variables in first-order predicates that match the type signature of the predicate. For binary predicates, this results in grounded networks which are often nearly quadratic in the number of entities in the knowledge base. Scaling Markov Logic to large knowledge bases with millions of entities (and hundreds of millions of facts) thus remains all but straightforward. Recently, the Tuffy [103] engine (see Section 4) has been addressing the issue of scaling up Markov Logic by coupling Alchemy[19] with a relational back-end, and by replacing the grounding procedure of Alchemy with a more efficient bottom-up grounding technique.

MAP-Inference. More recently, stochastic ways of addressing inference over a combination of deterministic (hard) and probabilistic (soft) dependencies has been addressed also in the context of Markov Logic. Maximum-a-Posteriori (MAP) inference [133] (based on the stochastic weighted Max-SAT solver MaxWalkSAT [74]) and MC-SAT [114] (based on slice sampling [33]) are two approximation algorithms for propositional and probabilistic inference in Markov Logic, respectively. Using a log-linear model for generating the factors of grounded formulas, MAP-inference can be shown to directly correspond to an execution of MaxWalkSAT over a Markov Logic network [119].

MC-SAT. Hard constraints may introduce isolated regions of states which cannot easily be overcome by a Gibbs sampler (i.e, by just flipping one variable at a time). MC-SAT [113] thus introduces auxiliary variables which provide the sampler with the ability a "jump" into another (otherwise disconnected) region with some probability. Experimentally, MC-SAT has been shown to outperform Gibbs sampling and simulated tempering by a significant margin, particularly when deterministic dependencies are present. However, allowing arbitrary constraints as hard rules may lead to the formulation of unsatisfiable constraints, which either renders the knowledge base inconsistent (if there is no solution at all) or empty (if the only solution is to set all facts to false). Satisfiability checks, which includes checking whether a derived fact is false in all the possible worlds and thus has a probability of exactly 0, cannot be approximated and thus remain an NP-hard problem.

[19] `http://alchemy.cs.washington.edu/`

Constrained Conditional Models. Another framework which combines (first-order) logical constraints and probabilistic inference is given by Constrained Conditional Models [88] (CCMs). Intuitively, constraints between input variables (observations) and output variables (labels) are encoded into linear weight vectors, which can be solved by Integer Linear Programming. Working with CCMs involves both learning weights for the model and efficient inference. CCMs allow for encoding Markov Random Fields, Hidden Markov Models and Conditional Random Fields [121]. They found strong applications in natural language processing, including tasks like co-reference resolution, semantic role labeling, and information extraction.

Probabilistic Datalog. Early probabilistic extensions to Datalog have been studied already in [44] and have later been refined to a number OWL concepts [104]. Although this approach already introduced a notion of lineage (coined "intensional query semantics"), the probabilistic computations are restricted to a class of rules which is guaranteed to provide independent subgoals (similar to the notion of safe plans or read-once functions in [30, 129]), where confidence computations can be propagated "upwards" the lineage tree using the inclusion-exclusion principle (aka. "sieve formula").

3.4 Programming Platforms for Probabilistic Inference

The "declarative-imperative" [64], a term coined in the context of the Berkeley Orders of Magnitude (BOOM) project[20], brings two seemingly contracting paradigms in data management to the point: how can we combine the power of an imperative programming language with the convenience of a declarative query language? In the following, we briefly highlight two imperative programming platforms for probabilistic inference: FACTORIE and Infer.NET.

FACTORIE[21] is a toolkit for deployable probabilistic modeling developed by the machine learning group at the University of Massachusetts Amherst [93]. It is based on the idea of using an imperative programming language (Scala) to define templates which generate factors between random variables, an approach coined *imperatively defined factor graphs*. Intuitively, when instantiated these templates form a factor graph, where all factors that have been instantiated from the same template also share the same parameters that were used to define the template. For inference, FACTORIE provides a variety of techniques based on MCMC, including Gibbs sampling. FACTORIE has been successfully applied to various inference tasks in natural language processing and information integration. Recently, FACTORIE has also been coupled with a relational back-end and thus potentially scales to probabilistic database settings with billions of variables [156].

[20] `http://boom.cs.berkeley.edu/`
[21] `http://code.google.com/p/factorie/`

Infer.NET[22] provided by Microsoft Research in Cambridge provides a rich programming language for modeling Bayesian inference tasks in graphical models and comes with an out-of-the-box selection of inference algorithms. It provides a built-in API for defining random variables (binary/multivariate-discrete or continuous), factors, message-passing operators, and other algebraic operators. It has been used in many machine-learning settings, with tasks involving classification or clustering, and in a wide variety of domains, including information retrieval, bio-informatics, epidemiology, vision, and many others.

3.5 Distributed Probabilistic Inference

Distribution bears the highest potential to scale-up rule-based reasoning and probabilistic inference, but still is fairly unexplored in the context of uncertain reasoning and probabilistic data management. Although distribution of course cannot tackle the asymptotical runtime issues inherently involved in these algorithms, it bears two key advantages:

- Storing a large data or knowledge base with billions of uncertain data objects in a distributed environment immediately allows for an increased *main-memory locality* of the data, which is a key for both efficient rule-based reasoning and probabilistic inference, with a majority of fine-grained, random-access-style data accesses.
- Running queries over a cluster of machines bears great potential for high-performance *parallel computations*, but clearly also poses major algorithmic challenges in terms of synchronizing these computations and the preservation of approximation guarantees (e.g., convergence guarantees for the MCMC-based sampling techniques).

MCDB. The MCDB [68, 159] project at IBM Almaden focuses on supporting Monte Carlo techniques for complex data analysis tasks directly within a database system. MCDB is one of the few database approaches to probabilistic data management that has specifically been adapted to Hadoop[23], a massively parallel, Map-Reduce-like [35] computing environment. MCDB focuses on analytical tasks over a broad range of user-defined stochastic models, e.g., risk analysis with complex analytical queries including grouping and aggregations [53]. Another, recent, application domain of MCDB is declarative information extraction [95].

Message Passing & Distributed Inference. Its iterative nature makes the Map-Reduce paradigm not well suitable for inference tasks, which inherently involve many fine-grained updates between states of objects that may be distributed across a compute cluster. For probabilistic inference, two main

[22] `http://research.microsoft.com/en-us/um/cambridge/projects/infernet/`
[23] `http://hadoop.apache.org/`

paradigms for distribution co-exist: the *shared memory* and the *distributed memory* model. In the shared-memory model, every processor has access to all the memory in the cluster; while for the distributed memory model, every processor only has a limited amount of local memory, and each processor can pass "messages" to other processors in the cluster. An alternative to the classic Message Passing Interface[24] (MPI) is the Internet Communications Engine[25] (ICE). Both are shipped as C++ libraries.

With the ResidualSplash [49] algorithm, the authors present a parallel belief propagation algorithm under the shard memory model, which is shown to achieve optimal runtime compared to a theoretical lower bound for parallel inference on chain graphical models. In their later DBRSplash [50] algorithm, the authors drop the shared memory model and consider parallel inference techniques in generic probabilistic graphical models, which are captured as *distributed factor graphs*. In this setting, a factor graph is distributed into a number of (disjoint or slightly overlapping) partitions, such that the number of partitions matches the number of processors available in the compute cluster. The objective of the partitioning function is to minimize the communication cost among nodes while ensuring load balance. Since computing an optimal partitioning under these constraints is NP-hard, an efficient (linear-time) approximation algorithm is devised as basis for the data partitioning. As for inference, a belief propagation algorithm is employed, with a local priority queue for incoming update messages at each processor. DBRSplash even reports a super-linear performance scale-up compared to a centralized setting. Moreover, GraphLab[26] [86] is a framework for deploying parallel (provably correct) machine learning algorithms. Unlike MapReduce, it focuses on more asynchronous communication protocols with different levels of sequential-consistency guarantees. In the *full consistency* model, during the execution of a function $f(v)$ on a vertex v, no other function is allowed to read or write data to any node in the scope (neighborhood) $S(v)$. In the *edge consistency* model, no other function is allowed to read or write data to an edge associated with v, while a function $f(v)$ is executed. Finally, in the weakest form of consistency model, the *vertex consistency* model, no other function is allowed to read or write data to the vertex v itself, while a function $f(v)$ is executed.

4 New trends: BayesStore, SPROUT, Tuffy, URDF

SQL-style query processing over uncertain relational data or, respectively, rule-based reasoning with uncertain RDF data is an emerging topic in the database, knowledge management and semantic web communities. Recent trends along these lines include moving away from strict relational data, lifting inference to higher-order constraints, and scaling-up inference for graphical models which combine first-order logic and probabilistic graphical models (in particular Markov Logic). Extracting structured data from the Web is an excellent

[24] http://www.mcs.anl.gov/research/projects/mpich2/
[25] http://www.zeroc.com/ice.html
[26] http://www.graphlab.ml.cmu.edu/

showcase—and likely one of the biggest challenges—for the scalable management of uncertain data we have seen so far. In the following, we thus briefly highlight a few projects, each of which devises exciting directions that could help making the management of uncertain RDF data scalable to billions of triples.

BayesStore. Based on a native extension to the PostgreSQL database system, the BayesStore [151] project at Berkeley aims to bridge the gap between statistical models induced by the data and the uncertainty model supported by the probabilistic database. BayesStore supports statistical models, evidence data, and inference algorithms directly as first-class citizens inside a database management system (DBMS). It combines a probabilistic database system (PDBMS) for relational data with a declarative first-order extension to Bayesian Nets, which allows for capturing complex correlation patterns in the PDBMS in a compact way. Using a combination of propositional and first-order factors, BayesStore supports probabilistic inference for all common database operations, including selection, projection, and joins. Recent extensions to BayesStore include the investigation of probabilistic-declarative information extraction techniques by using Conditional Random Fields (CRFs) for extraction and by implementing the Viterbi algorithm for efficient inference in the CRF directly via SQL [150].

SPROUT. Using the MayBMS probabilistic database system (see Section 3.1) as basis, the SPROUT (for "Scalable PROcessing on Uncertain Tables") [106] project at Oxford University aims at the scalable processing of queries over uncertain data. A particular focus of the project lies on tractable classes of queries for which exact probabilistic inference can be done in polynomial time. Similarly to the notion of safe plans [30], a probabilistic query is tractable if it has a hierarchical structure and (in the case of SPROUT) can be decomposed into a binary decision diagram in polynomial time. [107], on the other hand, investigates the decision diagrams for approximate inference for queries where exact probabilistic inference is intractable.

Tuffy. Based on the observation that Markov Logic does not easily scale to real-world datasets with millions of data objects, the Tuffy [103] at the University of Wisconsin-Madison investigates pushing Markov Logic Networks (MLNs) directly into a relational database system (RDBMS). In contrast to the open-world grounding strategy followed by the MLN implementation Alchemy, the authors pursue a more efficient bottom-up, closed-world grounding strategy of first-order rules through iterative SQL statements, which is fully supported by the DBMS optimizer. Focusing on MAP-inference, Tuffy provides a partitioning strategy for the search (inference) phase by splitting the grounded network into a number of independent components, which allows for parallel inference and an exponentially faster search compared to running the search on the global problem. For a given classification benchmark, Tuffy (consuming only 15MB of RAM) was reported to produce much better result quality within minutes than Alchemy (using 2.8GB of RAM) even after days of running.

URDF. In probabilistic databases, SQL is employed as means for formulating queries and for defining views. While SQL queries and schema-level constraints generally yield "hard" Boolean constraints among tuples, they lack the notion of "soft" dependencies among data items. The URDF [144] project developed at the Max Planck Institute for Informatics specifically focuses on weighted Horn clauses as soft rules and mutual-exclusion constraints as hard rules. Unlike Markov Logic, URDF follows a Datalog-style, deductive grounding strategy for soft rules in the form of Horn clauses, which typically results in a much smaller grounded network size than for Markov Logic. While URDF originally employed a Max-SAT solver, which had been tailored to a specific class of mutual-exclusion hard rules, URDF currently also employs probabilistic models based on deductive grounding and lineage. Moreover, URDF explores several directions for managing large amounts of web-extracted RDF data in a declarative way, including temporal reasoning extensions [39, 152], as well as inductively learning soft inference rules automatically from a given knowledge base. As exact inference for this class of queries is intractable, URDF investigates various approximation techniques based on MCMC (Gibbs sampling) for approximate inference.

References

1. RDF Primer & RDF Schema (W3C Rec.2004-02-10). `http://www.w3.org/TR/rdf-primer/`, `http://www.w3.org/TR/rdf-schema/`.
2. D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009.
3. D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, editors, *VLDB*, pages 411–422. ACM, 2007.
4. K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a self-organizing structured P2P system. *SIGMOD Rec.*, 32:29–33, September 2003.
5. K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. Van Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *The Semantic Web – ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 107–121. Springer Berlin / Heidelberg, 2004.
6. S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theor. Comput. Sci.*, 78(1):159–187, 1991.
7. G. Antoniou and F. van Harmelen. *A Semantic Web Primer (Cooperative Information Systems)*. The MIT Press, April 2004.
8. L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, pages 1479–1480, 2007.
9. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, editors, *WWW*, pages 41–50. ACM, 2010.

10. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: a nucleus for a web of open data. In *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, ISWC'07/ASWC'07, pages 722–735, 2007.
11. S. Auer, A.-D. N. Ngomo, and J. Lehmann. Introduction to linked data. In *Reasoning Web. Semantic Technologies for the Web of Data*, volume 6848 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011.
12. C. Beeri and R. Ramakrishnan. On the power of magic. *J. Log. Program.*, 10(1/2/3/4):255–299, 1991.
13. O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, pages 953–964, 2006.
14. T. Berners-Lee. Linked Data - Design Issues, 2006. http://www.w3.org/DesignIssues/LinkedData.html.
15. C. Bizer, T. Heath, and T. Berners-Lee. Linked Data – The Story So Far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.
16. J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Ré, and D. Suciu. MystiQ: a system for finding more answers by using probabilities. In *SIGMOD*, pages 891–893, 2005.
17. P. Bouquet, C. Ghidini, and L. Serafini. Querying the Web of Data: A Formal Approach. In A. Gmez-Prez, Y. Yu, and Y. Ding, editors, *The Semantic Web*, volume 5926 of *Lecture Notes in Computer Science*, pages 291–305. Springer Berlin / Heidelberg, 2009.
18. H. C. Bravo and R. Ramakrishnan. Optimizing MPF queries: decision support and probabilistic inference. In *SIGMOD*, pages 701–712, 2007.
19. P. Buitelaar, T. Eigner, and T. Declerck. OntoSelect: A Dynamic Ontology Library with Support for Ontology Selection. In *In Proceedings of the Demo Session at the International Semantic Web Conference*, 2004.
20. M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 650–657, 2004.
21. M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *Proceedings of the 4th International Workshop on Grid Computing*, GRID '03, pages 184–, 2003.
22. J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs. *Journal of Web Semantics*, 3:247–267, December 2005.
23. J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the Semantic Web recommendations. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *WWW (Alternate Track Papers & Posters)*, pages 74–83. ACM, 2004.
24. R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In *VLDB*, pages 71–81. Morgan Kaufmann, 1987.
25. H. Chen, Y. Wang, H. Wang, Y. Mao, J. Tang, C. Zhou, A. Yin, and Z. Wu. Towards a Semantic Web of relational databases: A practical semantic toolkit and an in-use case from traditional Chinese medicine. In *Fifth International Semantic Web Conference (ISWC)*, pages 750–763. Springer, 2006.
26. G. Cheng and Y. Qu. Searching Linked Objects with Falcons: Approach, Implementation and Evaluation. *Int. J. Semantic Web Inf. Syst.*, 5(3):49–70, 2009.
27. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-Å. Larson, and B. C. Ooi, editors, *VLDB*, pages 1216–1227. ACM, 2005.

28. K. L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
29. I. F. Cruz, V. Kashyap, S. Decker, and R. Eckstein, editors. *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases, Co-located with VLDB 2003, Humboldt-Universität, Berlin, Germany, September 7-8, 2003*, 2003.
30. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875, 2004.
31. N. Dalvi and D. Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *PODS Conference*, pages 293–302, 2007.
32. N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.
33. P. Damlen, J. Wakefield, and S. Walker. Gibbs sampling for Bayesian non-conjugate and hierarchical models by using auxiliary variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(2):331–344, 1999.
34. M. d'Aquin, C. Baldassarre, L. Gridinoc, S. Angeletou, M. Sabou, and E. Motta. Characterizing Knowledge on the Semantic Web with Watson. In *EON*, pages 1–10, 2007.
35. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
36. R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artif. Intell.*, 113(1-2):41–85, 1999.
37. L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs. Swoogle: a search and metadata engine for the semantic web. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 652–659, 2004.
38. Y. Ding, Y. Sun, B. Chen, K. Borner, L. Ding, D. Wild, M. Wu, D. DiFranzo, A. G. Fuenzalida, D. Li, S. Milojevic, S. Chen, M. Sankaranarayanan, and I. Toma. Semantic web portal: a platform for better browsing and visualizing semantic data. In *Proceedings of the 6th international conference on Active media technology*, AMT'10, pages 448–460, 2010.
39. M. Dylla, M. Sozio, and M. Theobald. Resolving temporal conflicts in inconsistent rdf knowledge bases. In *BTW*, pages 474–493, 2011.
40. O. Erling and I. Mikhailov. Towards web-scale rdf. http://virtuoso.openlinksw.com/whitepapers/Web-Scale
41. O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. In T. Pellegrini, S. Auer, K. Tochtermann, and S. Schaffert, editors, *Networked Knowledge - Networked Media*, volume 221 of *Studies in Computational Intelligence*, pages 7–24. Springer Berlin / Heidelberg, 2009.
42. G. H. L. Fletcher and P. W. Beck. Scalable indexing of RDF graphs for efficient join processing. In D. W.-L. Cheung, I.-Y. Song, W. W. Chu, X. Hu, and J. J. Lin, editors, *CIKM*, pages 1513–1516. ACM, 2009.
43. W. B. Frakes and R. A. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.
44. N. Fuhr. Probabilistic Datalog - a logic for powerful retrieval methods. In *SIGIR*, pages 282–290, 1995.
45. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming*, pages 1070–1080. MIT Press, 1988.
46. L. Getoor and B. Taskar. *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

47. W. Gilks, S. Richardson, and D. J. S. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman and Hall, 1996.
48. M. X. Goemans and D. P. Williamson. New 3/4-approximation algorithms for the maximum satisfiability problem. *SIAM J. Discrete Math.*, 7(4):656–666, 1994.
49. J. E. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics (AISTATS*, pages 177–184, 2009.
50. J. E. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *Uncertainty in Artificial Intelligence (UAI)*, pages 203–212, 2009.
51. O. Görlitz and S. Staab. *Federated Data Management and Query Optimization for Linked Open Data*, chapter 5, pages 109–137. Springer, 2011.
52. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
53. P. J. Haas, C. M. Jermaine, S. Arumugam, F. Xu, L. L. Perez, and R. Jampani. MCDB-R: Risk analysis in the database. *PVLDB*, 3(1):782–793, 2010.
54. P. Haase, T. Mathäß, and M. Ziller. An evaluation of approaches to federated query processing over linked data. In *Proceedings of the 6th International Conference on Semantic Systems*, I-SEMANTICS '10, pages 5:1–5:9, 2010.
55. P. Haase and Y. Wang. A decentralized infrastructure for query answering over distributed ontologies. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1351–1356, 2007.
56. S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In R. Volz, S. Decker, and I. F. Cruz, editors, *PSSS*, volume 89 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.
57. A. Harth. VisiNav: Visual Web Data Search and Navigation. In *Proceedings of the 20th International Conference on Database and Expert Systems Applications*, DEXA '09, pages 214–228, 2009.
58. A. Harth, A. Hogan, R. Delbru, J. Umbrich, S. O'Riain, and S. Decker. SWSE: Answers Before Links! In *Semantic Web Challenge*, 2007.
59. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. Data Summaries for On-Demand Queries over Linked Data. In *WWW'10*, pages 411–420, 2010.
60. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: a federated repository for querying graph structured data from the web. In *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, ISWC'07/ASWC'07, pages 211–224, 2007.
61. O. Hartig, C. Bizer, and J.-C. Freytag. Executing SPARQL Queries over the Web of Linked Data. In *The Semantic Web - ISWC 2009*, volume 5823, pages 293–309. Springer Berlin / Heidelberg, 2009.
62. O. Hartig and R. Heese. The sparql query graph model for query optimization. In *Proceedings of the 4th European conference on The Semantic Web: Research and Applications*, ESWC '07, pages 564–578, 2007.
63. O. Hartig and A. Langegger. A Database Perspective on Consuming Linked Data on the Web. *Datenbank-Spektrum*, 10(2):57–66, 2010.
64. J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.
65. A. Hogan, A. Harth, and S. Decker. ReConRank: A Scalable Ranking Method for Semantic Web Data with Context. In *In 2nd Workshop on Scalable Semantic Web Knowledge Base Systems*, 2006.

66. J. Huang, L. Antova, C. Koch, and D. Olteanu. MayBMS: a probabilistic database management system. In *SIGMOD*, pages 1071–1074, 2009.
67. T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.
68. R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. MCDB: a Monte Carlo approach to managing uncertain data. In J. T.-L. Wang, editor, *SIGMOD*, pages 687–700. ACM, 2008.
69. B. Jaumard and B. Simeone. On the complexity of the maximum satisfiability problem for Horn formulas. *Information Processing Letters*, 26(1):1 – 4, 1987.
70. A. Jha, V. Rastogi, and D. Suciu. Query evaluation with soft-key constraints. In *PODS*, pages 119–128, 2008.
71. B. Kanagal and A. Deshpande. Lineage processing over correlated probabilistic databases. In *SIGMOD*, pages 675–686, 2010.
72. P. C. Kanellakis and S. A. Smolka. CCS expressions finite state processes, and three problems of equivalence. *Inf. Comput.*, 86:43–68, May 1990.
73. R. M. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *FOCS*, pages 56–64, 1983.
74. H. Kautz, B. Selman, and Y. Jiang. A general stochastic approach to solving problems with hard and soft constraints. In *The Satisfiability Problem: Theory and Applications*, pages 573–586. American Mathematical Society, 1996.
75. C. Koch. A compositional query algebra for second-order logic and uncertain databases. In *ICDT*, pages 127–140, 2009.
76. D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32:422–469, December 2000.
77. R. A. Kowalski and D. Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.
78. G. Ladwig and T. Tran. Linked Data Query Processing Strategies. In *International Semantic Web Conference (ISWC'10)*, pages 453–469, 2010.
79. A. Langegger, W. Wöß, and M. Blöchl. A semantic web middleware for virtual data integration on the web. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 493–507, 2008.
80. J. J. Levandoski and M. F. Mokbel. RDF data-centric storage. In *ICWS*, pages 911–918. IEEE, 2009.
81. J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1):502–513, 2009.
82. S. Liang, P. Fodor, H. Wan, and M. Kifer. OpenRuleBench: an analysis of the performance of rule engines. In *WWW*, pages 601–610. ACM, 2009.
83. E. Liarou, S. Idreos, and M. Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In *In International Semantic Web Conference (ISWC'06)*, pages 399–413. Springer Heidelberg, 2006.
84. B. Liu and B. Hu. Path queries based RDF index. In *SKG*, page 91. IEEE Computer Society, 2005.
85. B. Liu and B. Hu. HPRD: A high performance RDF database. In K. Li, C. R. Jesshope, H. Jin, and J.-L. Gaudiot, editors, *NPC*, volume 4672 of *Lecture Notes in Computer Science*, pages 364–374. Springer, 2007.
86. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.
87. T. Lukasiewicz. Probabilistic description logic programs. *Int. J. Approx. Reasoning*, 45(2):288–307, 2007.

88. N. R. M. Chang, L. Ratinov and D. Roth. Learning and inference with constraints. In *AAAI*, 2008.
89. A. Maduko, K. Anyanwu, A. Sheth, and P. Schliekelman. Graph summaries for subgraph frequency estimation. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 508–523, 2008.
90. A. Maduko, K. Anyanwu, A. P. Sheth, and P. Schliekelman. Estimating the cardinality of RDF graph patterns. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *WWW*, pages 1233–1234. ACM, 2007.
91. A. Maduko, K. Anyanwu, A. P. Sheth, and P. Schliekelman. Graph summaries for subgraph frequency estimation. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 508–523. Springer, 2008.
92. A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura. An indexing scheme for RDF and RDF schema based on suffix arrays. In Cruz et al. [29], pages 151–168.
93. A. McCallum, K. Schultz, and S. Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *NIPS*, 2009.
94. A. O. Mendelzon and T. Milo. Formal models of Web queries. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '97, pages 134–143, 1997.
95. E. Michelakis, R. Krishnamurthy, P. J. Haas, and S. Vaithyanathan. Uncertainty management in rule-based information extraction systems. In *SIGMOD*, pages 101–114, 2009.
96. M. Mutsuzaki, M. Theobald, A. de Keijzer, J. Widom, P. Agrawal, O. Benjelloun, A. D. Sarma, R. Murthy, and T. Sugihara. Trio-One: Layering uncertainty and lineage on a conventional DBMS (demo). In *CIDR*, pages 269–274, 2007.
97. N. Nakashole, M. Theobald, and G. Weikum. Scalable knowledge harvesting with high precision and high recall. In *WSDM*, pages 227–236, 2011.
98. W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 604–615, New York, NY, USA, 2002. ACM.
99. T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.
100. T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *SIGMOD Conference*, pages 627–640. ACM, 2009.
101. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.
102. I. Niemel and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning*. Springer, 1997.
103. F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: scaling up statistical inference in Markov logic networks using an RDBMS. Technical report, University of Wisconsin-Madison, 2010.
104. H. Nottelmann and N. Fuhr. Adding probabilities and rules to OWL lite subsets based on probabilistic Datalog. *Int. Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 14(1):17–41, 2006.
105. P. Obermeier and L. Nixon. A Cost Model for Querying Distributed RDF-Repositories with SPARQL. In *Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, 2008.

106. D. Olteanu, J. Huang, and C. Koch. SPROUT: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, pages 640–651. IEEE, 2009.
107. D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, pages 145–156, 2010.
108. E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tummarello. Sindice.com: a document-oriented lookup index for open linked data. *Int. J. Metadata Semant. Ontologies*, 3:37–52, November 2008.
109. L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999.
110. R. Palma and P. Haase. Oyster - Sharing and Re-using Ontologies in a Peer-to-Peer Community. In *The Semantic Web – ISWC 2005*, Lecture Notes in Computer Science, pages 1059–1062. Springer Berlin / Heidelberg, 2005.
111. J. Z. Pan, E. Thomas, and D. Sleeman. Ontosearch2: Searching and querying web ontologies. In *In Proc. of the IADIS International Conference*, pages 211–218, 2006.
112. C. Patel, K. Supekar, Y. Lee, and E. K. Park. OntoKhoj: a semantic web portal for ontology searching, ranking and classification. In *Proceedings of the 5th ACM international workshop on Web information and data management*, WIDM '03, pages 58–61, 2003.
113. H. Poon and P. Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *AAAI*. AAAI Press, 2006.
114. H. Poon, P. Domingos, and M. Sumner. A general method for reducing the complexity of relational inference and its application to MCMC. In *AAAI*, pages 1075–1080, 2008.
115. B. Quilitz and U. Leser. Querying distributed RDF data sources with SPARQL. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 524–538, 2008.
116. C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.
117. C. Re and D. Suciu. Managing probabilistic data with MystiQ: The can-do, the could-do, and the can't-do. In *SUM*, pages 5–18, 2008.
118. M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2), 2006.
119. S. Riedel. Cutting plane MAP inference for Markov Logic. In *International Workshop on Statistical Relational Learning (SRL)*, 2009.
120. D. Roth. On the hardness of approximate reasoning. *Artif. Intell.*, 82:273–302, April 1996.
121. D. Roth and W. Yih. Integer linear programming inference for conditional random fields. In *Proc. of the International Conference on Machine Learning (ICML)*, pages 737–744, 2005.
122. S. Sakr and G. Al-Naymat. Relational processing of rdf queries: a survey. *SIGMOD Record*, 38(4):23–28, 2009.
123. A. D. Sarma, O. Benjelloun, A. Y. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, page 7, 2006.
124. A. D. Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, pages 1023–1032, 2008.
125. A. D. Sarma, M. Theobald, and J. Widom. LIVE: A lineage-supported versioned DBMS. In *SSDBM*, pages 416–433, 2010.

126. S. Schenk and S. Staab. Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the Web. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 585–594, 2008.
127. P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, pages 596–605, 2007.
128. P. Sen, A. Deshpande, and L. Getoor. PrDB: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5):1065–1090, 2009.
129. P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. *PVLDB*, 3(1):1068–1079, 2010.
130. L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.
131. S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. E. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *SIGMOD*, pages 1239–1242, 2008.
132. S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. E. Hambrusch, J. Neville, and R. Cheng. Database support for probabilistic attributes and tuples. In *ICDE*, pages 1053–1061, 2008.
133. P. Singla and P. Domingos. Memory-efficient inference in relational domains. In *AAAI*, 2006.
134. M. A. Soliman, I. F. Ilyas, and K. C. Chang. URank: formulation and efficient evaluation of top-k queries in uncertain databases. In *SIGMOD*, pages 1082–1084, 2007.
135. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In J. Huai, R. Chen, H.-W. Hon, Y. Liu, W.-Y. Ma, A. Tomkins, and X. Zhang, editors, *WWW*, pages 595–604. ACM, 2008.
136. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 595–604, 2008.
137. I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11:17–32, February 2003.
138. U. Straccia. Reasoning Web. chapter Managing Uncertainty and Vagueness in Description Logics, Logic Programs and Description Logic Programs, pages 54–103. Springer-Verlag, Berlin, Heidelberg, 2008.
139. H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed RDF repositories. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 631–639, 2004.
140. F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.
141. F. M. Suchanek, M. Sozio, and G. Weikum. SOFIE: a self-organizing framework for information extraction. In *WWW*, pages 631–640, 2009.
142. A. W. Systeme, G. Gottlob, A. Voronkov, E. Dantsin, E. Dantsin, T. Eiter, and T. Eiter. Complexity and expressive power of logic programming, 1999.
143. G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory Pract. Log. Program.*, 8:129–165, March 2008.

144. M. Theobald, M. Sozio, F. Suchanek, and N. Nakashole. URDF: Efficient reasoning in uncertain RDF knowledge bases with soft and hard rules. Technical Report MPII20105-002, Max Planck Institute Informatics (MPI-INF), 2010.
145. Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of RDF/S stores. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 685–701. Springer, 2005.
146. T. Tran, P. Haase, and R. Studer. Semantic Search – Using Graph-Structured Semantic Models for Supporting the Search Process. In *Proceedings of the 17th International Conference on Conceptual Structures: Conceptual Structures: Leveraging Semantic Technologies*, ICCS '09, pages 48–65, 2009.
147. T. Tran, H. Wang, and P. Haase. Hermes: Data Web search on a pay-as-you-go integration infrastructure. *Web Semant.*, 7:189–203, September 2009.
148. G. Tummarello, R. Cyganiak, M. Catasta, S. Danielczyk, R. Delbru, and S. Decker. Sig.ma: live views on the web of data. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 1301–1304, 2010.
149. O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. In *AAAI*, pages 1465–1470. AAAI Press, 2007.
150. D. Z. Wang, E. Michelakis, M. J. Franklin, M. N. Garofalakis, and J. M. Hellerstein. Probabilistic declarative information extraction. In *ICDE*, pages 173–176, 2010.
151. D. Z. Wang, E. Michelakis, M. N. Garofalakis, and J. M. Hellerstein. BayesStore: managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 1(1):340–351, 2008.
152. Y. Wang, M. Yahya, and M. Theobald. Time-aware reasoning in uncertain knowledge bases. In *Workshop on Management of Uncertain Data (MUD)*, 2010.
153. D. S. Warren. Memoing for logic programs. *Commun. ACM*, 35:93–111, 1992.
154. W. Wei, J. Erenrich, and B. Selman. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, pages 670–676, 2004.
155. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for Semantic Web data management. *PVLDB*, 1(1):1008–1019, 2008.
156. M. L. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and mcmc. *PVLDB*, 3(1):794–804, 2010.
157. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *First International Workshop on Semantic Web and Databases (SWDB '03)*, pages 131–150, 2003.
158. K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In Cruz et al. [29], pages 131–150.
159. F. Xu, K. S. Beyer, V. Ercegovac, P. J. Haas, and E. J. Shekita. E = MC^3: managing uncertain enterprise data in a cluster-computing environment. In *SIGMOD*, pages 441–454, 2009.
160. M. Zhou and Y. Wu. XML-based RDF data management for efficient query processing. In X. L. Dong and F. Naumann, editors, *WebDB*, 2010.