

Chapter 12

Web Data Management



The World Wide Web (“WWW” or “web” for short) has become a major repository of data and documents. Although measurements differ and change, the web has grown at a phenomenal rate.¹ Besides its size, the web is very dynamic and changes rapidly. For all practical purposes, the web represents a very large, dynamic, and distributed data store and there are the obvious distributed data management issues in accessing web data.

The web, in its present form, can be viewed as two distinct yet related components. The first of these components is what is known as the *publicly indexable web* (PIW) that is composed of all static (and cross-linked) web pages that exist on web servers. These can be easily searched and indexed. The other component, which is known as the *deep web* (or the *hidden web*), is composed of a huge number of databases that encapsulate the data, hiding it from the outside world. The data in the hidden web are usually retrieved by means of search interfaces where the user enters a query that is passed to the database server, and the results are returned to the user as a dynamically generated web page. A portion of the deep web has come to be known as the “dark web,” which consists of encrypted data and requires a particular browser such as Tor to access.

The difference between the PIW and the hidden web is basically in the way they are handled for searching and/or querying. Searching the PIW depends mainly on crawling its pages using the link structure between them, indexing the crawled pages, and then searching the indexed data (as we discuss at length in Sect. 12.2). This may be either through the well-known keyword search or via question answering (QA) systems (Sect. 12.4). It is not possible to apply this approach to the hidden web directly since it is not possible to crawl and index those data (the techniques for searching the hidden web are discussed in Sect. 12.5).

The original version of this chapter was revised. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-26253-2_13

¹See <http://www.worldwidewebsite.com/>.

Research on web data management has followed different threads in two separate but overlapping communities. Most of the earlier work in the web search and information retrieval community focused on keyword search and search engines. Subsequent work in this community focused on QA systems. The work in the database community focused on declarative querying of web data. There is an emerging trend that combines search/browse mode of access with declarative querying, but this work has not yet reached its full potential. In the 2000s, XML emerged as an important data format for representing and integrating data on the web. Thus, XML data management was a topic of significant interest. Although XML is still important in a number of application areas, its use in web data management has waned, mostly due to its perceived complexity. More recently, RDF has emerged as a common representation for Web data representation and integration.

The result of these different threads of development is that there is little in the way of a unifying architecture or framework for discussing web data management, and the different lines of research have to be considered somewhat separately. Furthermore, the full coverage of all the web-related topics requires far deeper and far more extensive treatment than is possible within a chapter. Therefore, we focus on issues that are directly related to data management.

We start by discussing how web data can be modeled as a graph. Both the structure of this graph and its management are important. This is discussed in Sect. 12.1. Web search is discussed in Sect. 12.2 and web querying is covered in Sect. 12.3. Section 12.4 summarizes question answering systems, and searching and querying the deep/hidden web is covered in Sect. 12.5. We then discuss web data integration in Sect. 12.6, focusing both on the fundamental problems and some of the representation approaches (e.g., web tables, XML, and RDF) that can assist with the task.

12.1 Web Graph Management

The web consists of “pages” that are connected by hyperlinks, and this structure can be modeled as a directed graph that reflects the hyperlink structure. In this graph, commonly referred to as the *web graph*, static HTML web pages are the vertices and the links between the pages are represented as directed edges. The characteristics of the web graph is important for studying data management issues since the graph structure is exploited in web search, categorization and classification of web content, and other web-related tasks. In addition, RDF representation that we discuss in Sect. 12.6.2.2 formalizes the web graph using a particular notation. The important characteristics of the web graph are the following:

- (a) It is quite volatile. We already discussed the speed with which the graph is growing. In addition, a significant proportion of the web pages experience frequent updates.
- (b) It is sparse. A graph is considered sparse if its average degree (i.e., the average of the degrees of all of its vertices) is less than the number of vertices. This

means that each vertex of the graph has a limited number of neighbors, even if the vertices are in general connected. The sparseness of the web graph implies an interesting graph structure that we discuss shortly.

- (c) It is “self-organizing.” The web contains a number of communities, each of which consists of a set of pages that focus on a particular topic. These communities get organized on their own without any “centralized control,” and give rise to the particular subgraphs in the web graph.
- (d) It is a “small-world graph.” This property is related to sparseness—each node in the graph may not have many neighbors (i.e., its degree may be small), but many nodes are connected through intermediaries. Small-world networks were first identified in social sciences where it was noted that many people who are strangers to each other are connected by intermediaries. This holds true in web graphs as well in terms of the connectedness of the graph.
- (e) It is a power law graph. The in- and out-degree distributions of the web graph follow power law distributions. This means that the probability that a vertex has in- (out-) degree i is proportional to $1/i^\alpha$ for some $\alpha > 1$. The value of α is about 2.1 for in-degree and about 7.2 for out-degree.

This brings us to a discussion of the structure of the web graph, which has a “bowtie” shape (Fig. 12.1). It has a strongly connected component (the knot in the middle) in which there is a path between each pair of pages. The numbers we give below are from a study in 2000; while these numbers have possibly changed, the structure depicted in the figure has persisted. Readers should treat numbers as indicative of relative size and not as absolute values. The strongly connected component (SCC) accounts for about 28% of the web pages. A further 21% of the pages constitute the “IN” component from which there are paths to pages in SCC, but to which no paths exist from pages in SCC. Symmetrically, “OUT” component has pages to which paths exist from pages in SCC but not vice versa, and these also constitute 21% of the pages. What is referred to as “tendrils” consist of pages that cannot be reached from SCC and from which SCC pages cannot be reached

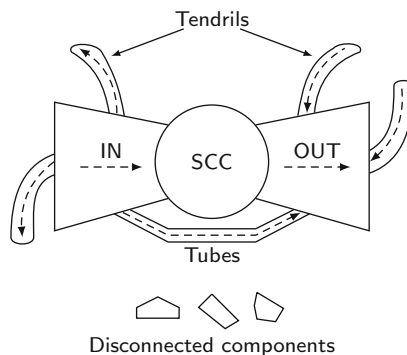


Fig. 12.1 The structure of the web as a bowtie (based on [Kumar et al. 2000])

either. These constitute about 22% of the web pages. These are pages that have not yet been “discovered” and have not yet been connected to the better known parts of the web. Finally, there are disconnected components that have no links to/from anything except their own small communities. This makes up about 8% of the web. This structure is interesting in that it determines the results that one gets from web searches and from querying the web. Furthermore, this graph structure is different than many other graphs that are normally studied, requiring special algorithms and techniques for its management.

12.2 Web Search

Web search involves finding “all” the web pages that are relevant (i.e., have content related) to keyword(s) that a user specifies. Naturally, it is not possible to find all the pages, or even to know if one has retrieved all the pages; thus the search is performed on a database of web pages that have been collected and indexed. Since there are usually multiple pages that are relevant to a query, these pages are presented to the user in ranked order of relevance as determined by the search engine.

The abstract architecture of a generic search engine is shown in Fig. 12.2. We discuss the components of this architecture in some detail.

In every search engine the *crawler* plays one of the most crucial roles. A crawler is a program used by a search engine to scan the web on its behalf and collect data about web pages. A crawler is given a starting set of pages—more accurately, it is given a set of Uniform Resource Locators (URLs) that identify these pages. The crawler retrieves and parses the page corresponding to that URL, extracts any

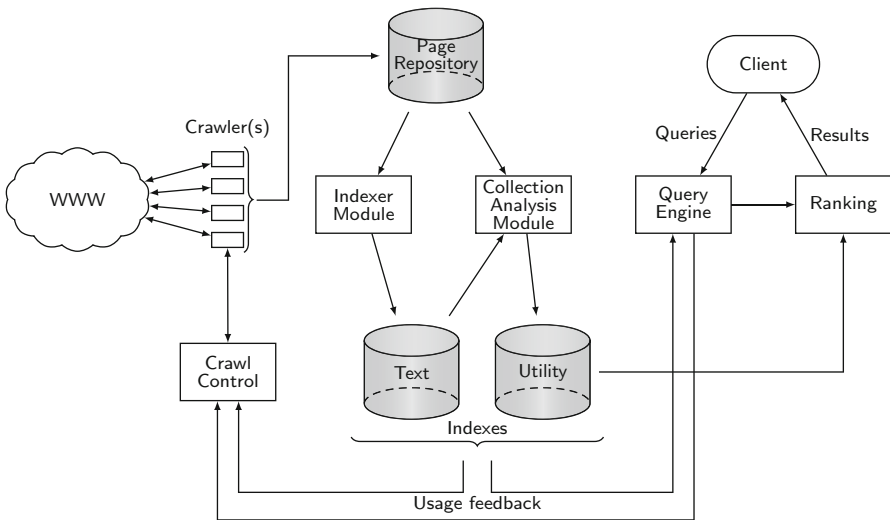


Fig. 12.2 Search engine architecture (based on [Arasu et al. 2001])

URLs in it, and adds these URLs to a queue. In the next cycle, the crawler extracts a URL from the queue (based on some order) and retrieves the corresponding page. This process is repeated until the crawler stops. A control module is responsible for deciding which URLs should be visited next. The retrieved pages are stored in a page repository. Section 12.2.1 examines crawling operations in more detail.

The *indexer module* is responsible for constructing indexes on the pages that have been downloaded by the crawler. While many different indexes can be built, the two most common ones are *text indexes* and *link indexes*. In order to construct a text index, the indexer module constructs a large “lookup table” that can provide all the URLs that point to the pages where a given word occurs. A link index describes the link structure of the web and provides information on the in-link and out-link state of pages. Section 12.2.2 explains current indexing technology and concentrates on ways indexes can be efficiently stored.

The *ranking module* is responsible for sorting a large number of results so that those that are considered to be most relevant to the user’s search are presented first. The problem of ranking has drawn increased interest in order to go beyond traditional information retrieval (IR) techniques to address the special characteristics of the web—web queries are usually small and they are executed over a vast amount of data. Section 12.2.3 introduces algorithms for ranking and describes approaches that exploit the link structure of the web to obtain improved ranking results.

12.2.1 Web Crawling

As indicated above, a crawler scans the web on behalf of a search engine to extract information about the visited web pages. Given the size of the web, the changing nature of web pages, and the limited computing and storage capabilities of crawlers, it is impossible to crawl the entire web. Thus, a crawler must be designed to visit “most important” pages before others. The issue, then, is to visit the pages in some ranked order of importance.

There are a number of issues that need to be addressed in designing a crawler. Since the primary goal is to access more important pages before others, there needs to be some way of determining the importance of a page. This can be done by means of a measure that reflects the importance of a given page. These measures can be static, such that the importance of a page is determined independent of retrieval queries that will run against it, or dynamic in that they take the queries into consideration. Examples of static measures are those that determine the importance of a page P_i with respect to the number of pages that point to P_i (referred to as *backlink*), or those that additionally take into account the importance of the backlink pages as is done in the popular PageRank metric that is used by Google and others. A possible dynamic measure may be one that calculates the importance of a page P_i with respect its textual similarity to the query that is being evaluated using some of the well-known information retrieval similarity measures.

We had introduced PageRank in the Chap. 10 (Example 10.4). Recall that the PageRank of a page P_i , denoted $PR(P_i)$, is simply the normalized sum of the PageRank of all P_i 's backlink pages (denoted as B_{P_i}) where the normalization for each $P_j \in B_{P_i}$ is over all of P_j 's forward links F_{P_j} :

$$PR(P_i) = \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}$$

Recall also that this formula calculates the rank of a page based on the backlinks, but normalizes the contribution of each backlinking page P_j using the number of forward links that P_j has. The idea here is that it is more important to be pointed at by pages conservatively link to other pages than by those who link to others indiscriminately, but the “contribution” of a link from such a page needs to be normalized over all the pages that it points to.

A second issue is how the crawler chooses the next page to visit once it has crawled a particular page. As noted earlier, the crawler maintains a queue in which it stores the URLs for the pages that it discovers as it analyzes each page. Thus, the issue is one of ordering the URLs in this queue. A number of strategies are possible. One possibility is to visit the URLs in the order in which they were discovered; this is referred to as the *breadth-first approach*. Another alternative is to use random ordering whereby the crawler chooses a URL randomly from among those that are in its queue of unvisited pages. Other alternatives are to use metrics that combine ordering with importance ranking discussed above, such as backlink counts or PageRank.

Let us discuss how PageRank can be used for this purpose. A slight revision is required to the PageRank formula given above. We are now modeling a random surfer: when landed on a page P , a random surfer is likely to choose one of the URLs on this page as the next one to visit with some (equal) probability d or will jump to a random page with probability $1-d$. Then the above formula for PageRank is revised as follows:

$$PR(P_i) = (1-d) + d \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}$$

The ordering of the URLs according to this formula allows the importance of a page to be incorporated into the order in which the corresponding page is visited. In some formulations, the first term is normalized with respect to the total number of pages in the web.

Example 12.1 Consider the web graph in Fig. 12.3 where each web page P_i is a vertex and there is a directed edge from P_i to P_j if P_i has a link to P_j . Assuming the commonly accepted value of $d = 0.85$, the PageRank of P_2 is $PR(P_2) = 0.15 + 0.85(\frac{PR(P_1)}{2} + \frac{PR(P_3)}{3})$. This is a recursive formula that is evaluated by initially assigning to each page equal PageRank values (in this case $\frac{1}{6}$ since there are 6

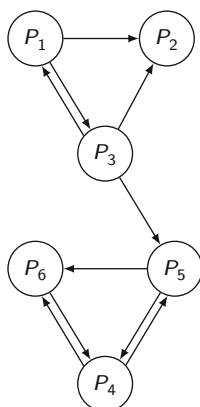


Fig. 12.3 Web graph representation for PageRank computation

pages) and iterating to compute each $PR(P_i)$ until a fixpoint is reached (i.e., the values no longer change). ♦

Since many web pages change over time, crawling is a continuous activity and pages need to be revisited. Instead of restarting from scratch each time, it is preferable to selectively revisit web pages and update the gathered information. Crawlers that follow this approach are called *incremental crawlers*. They ensure that the information in their repositories are as fresh as possible. Incremental crawlers can determine the pages that they revisit based on the change frequency of the pages or by sampling a number of pages. *Change frequency-based* approaches use an estimate of the change frequency of a page to determine how frequently it should be revisited. One might intuitively assume that pages with high change frequency should be visited more often, but this is not always true—any information extracted from a page that changes frequently is likely to become obsolete quickly, and it may be better to increase revisit interval to that page. It is also possible to develop an adaptive incremental crawler such that the crawling in one cycle is affected by the information collected in the previous cycle. *Sampling-based approaches* focus on web sites rather than individual web pages. A small number of pages from a web site are sampled to estimate how much change has happened at the site. Based on this sampling estimate, the crawler determines how frequently it should visit that site.

Some search engines specialize in searching pages belonging to a particular topic. These engines use crawlers optimized for the target topic, and are referred to as *focused crawlers*. A focused crawler ranks pages based on their relevance to the target topic, and uses them to determine which pages it should visit next. Classification techniques that are widely used in information retrieval are used in evaluating relevance; learning techniques are used to identify the topic of a given page. These techniques are beyond our scope, but a number of them have been developed for this purpose, such as naïve Bayes classifier, and its extensions, reinforcement learning, and others.

To achieve reasonable scale-up, crawling can be parallelized by running *parallel crawlers*. Any design for parallel crawlers must use schemes to minimize the overhead of parallelization. For instance, two crawlers running in parallel may download the same set of pages. Clearly, such overlap needs to be prevented through coordination of the crawlers' actions. One method of coordination uses a *central coordinator* to dynamically assign each crawler a set of pages to download. Another coordination scheme is to logically partition the web. Each crawler knows its partition, and there is no need for central coordination. This scheme is referred to as the *static assignment*.

12.2.2 Indexing

In order to efficiently search the crawled pages and the gathered information, a number of indexes are built as shown in Fig. 12.2. The two more important indexes are the *structure* (or *link*) *index* and a *text* (or *content*) *index*.

12.2.2.1 Structure Index

The structure index is based on the graph model that we discussed in Sect. 12.1, with the graph representing the structure of the crawled portion of the web. The efficient storage and retrieval of these pages is important and two techniques to address these issues were discussed in Sect. 12.1. The structure index can be used to obtain important information about the linkage of web pages such as information regarding the *neighborhood* of a page and the siblings of a page.

12.2.2.2 Text Index

The most important and mostly used index is the *text index*. Indexes to support text-based retrieval can be implemented using any of the access methods traditionally used to search over text document collections. Examples include *suffix arrays*, *inverted files* or *inverted indexes*, and *signature files*. Although a full treatment of all of these indexes is beyond our scope, we will discuss how inverted indexes are used in this context since these are the most popular text indexes.

An inverted index is a collection of inverted lists, where each list is associated with a particular word. In general, an inverted list for a given word is a list of document identifiers in which that particular word occurs. The location of the word on a particular page can also be saved as part of the inverted list. This information is usually needed in proximity queries and query result ranking. Search algorithms also often make use of additional information about the occurrence of terms in a web page. For example, terms occurring in bold face (within `` tags), in section headings (within `<H1>` or `<H2>` tags in HTML), or as anchor text might be weighted differently in the ranking algorithms.

In addition to the inverted list, many text indexes also keep a *lexicon*, which is a list of all terms that occur in the index. The lexicon can also contain some term-level statistics that can be used by ranking algorithms.

Constructing and maintaining an inverted index has three major difficulties:

1. In general, building an inverted index involves processing each page, reading all the words and storing the location of each word. In the end, the inverted files are written to disk. This process, while trivial for small and static collections, becomes hard to manage when dealing with a vast and nonstatic collection like the web.
2. The rapid change of the web poses a challenge for maintaining the “freshness” of the index. Although we argued in the previous section that incremental crawlers should be deployed to ensure freshness, periodic index rebuilding is still necessary because most incremental update techniques do not perform well when dealing with the large changes often observed between successive crawls.
3. Storage formats of inverted indexes must be carefully designed. There is a tradeoff between a performance gain through a compressed index that allows portions of the index to be cached in memory and the overhead of decompression at query time. Achieving the right balance becomes a major concern when dealing with web-scale collections.

Addressing these challenges and developing a highly scalable text index can be achieved by distributing the index by either building a *local inverted index* at each machine where the search engine runs or building a *global inverted index* that is then shared. We do not discuss these further, as the issues are similar to the distributed data and directory management issues we have already covered in previous chapters.

12.2.3 Ranking and Link Analysis

A typical search engine returns a large number of web pages that are expected to be relevant to a user query. However, these pages are likely to be different in terms of their quality and relevance. The user is not expected to browse through this large collection to find a high-quality page. Clearly, there is a need for algorithms to rank these pages such that higher quality web pages appear as part of the top results.

Link-based algorithms can be used to rank a collection of pages. To repeat what we discussed earlier, the intuition is that if a page P_j contains a link to page P_i , then it is likely that the authors of page P_j think that page P_i is of good quality. Thus, a page that has a large number of incoming links is likely of high quality, and hence the number of incoming links to a page can be used as a ranking criterion. This intuition is the basis of ranking algorithms, but, of course, each specific algorithm implements this intuition in a different way. We already discussed the PageRank algorithm, and it is used for ranking of results in addition to crawling. We will discuss an alternative algorithm called HITS to highlight different ways of approaching the issue.

HITS is also a link-based algorithm. It is based on identifying “authorities” and “hubs.” A good authority page receives a high rank. Hubs and authorities have a mutually reinforcing relationship: a good authority is a page that is linked to by many good hubs, and a good hub is a document that links to many authorities. Thus, a page pointed to by many hubs (a good authority page) is likely to be of high quality.

Let us start with a web graph, $G = (V, E)$, where V is the set of pages and E is the set of links among them. Each page P_i in V has a pair of nonnegative weights (a_{P_i}, h_{P_i}) that represent the authoritative and hub values of P_i respectively.

The authoritative and hub values are updated as follows. If a page P_i is pointed to by many good hubs, then a_{P_i} is increased to reflect all pages P_j that link to it (the notation $P_j \rightarrow P_i$ means that page P_j has a link to page P_i):

$$a_{P_i} = \sum_{\{P_j | P_j \rightarrow P_i\}} h_{P_j}$$

$$h_{P_i} = \sum_{\{P_j | P_j \rightarrow P_i\}} a_{P_j}$$

Thus, the authoritative value (hub value) of page P_i , is the sum of the hub values (authority values) of all the backlink pages to P_i .

12.2.4 Evaluation of Keyword Search

Keyword-based search engines are the most popular tools to search information on the web. They are simple, and one can specify fuzzy queries that may not have an exact answer, but may only be answered approximately by finding facts that are “similar” to the keywords. However, there are obvious limitations as to how much one can do by simple keyword search. The obvious limitation is that keyword search is not sufficiently powerful to express complex queries. This can be (partially) addressed by employing iterative queries where previous queries by the same user can be used as the context for the subsequent queries. A second limitation is that keyword search does not offer support for a global view of information on the web the way that database querying exploits database schema information. It can, of course, be argued that a schema is meaningless for web data, but the lack of an overall view of the data is an issue nevertheless. A third problem is that it is difficult to capture user’s intent by simple keyword search—errors in the choice of keywords may result in retrieving many irrelevant answers.

Category search addresses one of the problems of using keyword search, namely the lack of a global view of the web. Category search is also known as web directory, catalogs, yellow pages, and subject directories. There are a number of public web

directories available such as World Wide Web Virtual Library (<http://vlib.org>).² The web directory is a hierarchical taxonomy that classifies human knowledge. Although, the taxonomy is typically displayed as a tree, it is actually a directed acyclic graph since some categories are cross referenced.

If a category is identified as the target, then the web directory is a useful tool. However, not all web pages can be classified, so the user can use the directory for searching. Moreover, natural language processing cannot be 100% effective for categorizing web pages. We need to depend on human resource for judging the submitted pages, which may not be efficient or scalable. Finally, some pages change over time, so keeping the directory up-to-date involves significant overhead.

There have also been some attempts to involve multiple search engines in answering a query to improve recall and precision. A metasearcher is a web service that takes a given query from the user and sends it to multiple heterogeneous search engines. The metasearcher then collects the answers and returns a unified result to the user. It has the ability to sort the result by different attributes such as host, keyword, date, and popularity. Examples include Dogpile (<http://www.dogpile.com/>), MetaCrawler (<http://www.metacrawler.com/>), and IxQuick (<http://www.ixquick.com/>). Different metasearchers have different ways to unify results and translate the user query to the specific query languages of each search engines. The user can access a metasearcher through client software or a web page. Each search engine covers a smaller percentage of the web. The goal of a metasearcher is to cover more web pages than a single search engine by combining different search engines together.

12.3 Web Querying

Declarative querying and efficient execution of queries have been a major focus of database technology. It would be beneficial if the database techniques can be applied to the web. In this way, accessing the web can be treated, to a certain extent, similar to accessing a large database. We will discuss a number of the proposed approaches in this section.

There are difficulties in carrying over traditional database querying concepts to web data. Perhaps the most important difficulty is that database querying assumes the existence of a strict schema. As noted above, it is hard to argue that there is a schema for web data similar to databases.³ At best, the web data are *semistructured*—data may have some structure, but this may not be as rigid, regular, or complete as that of databases, so that different instances of the data may be similar

²A list of these libraries is given in https://en.wikipedia.org/wiki/List_of_web_directories.

³We are focusing on the “open” web here; deep web data may have a schema, but it is usually not accessible to users.

but not identical (there may be missing or additional attributes or differences in structure). There are, obviously, inherent difficulties in querying schema-less data.

A second issue is that the web is more than the semistructured data (and documents). The links that exist between web data entities (e.g., pages) are important and need to be considered. Similar to search that we discussed in the previous section, links may need to be followed and exploited in executing web queries. This requires links to be treated as first-class objects.

A third major difficulty is that there is no commonly accepted language, similar to SQL, for querying web data. As we noted in the previous section, keyword search has a very simple language, but this is not sufficient for richer querying of web data. Some consensus on the basic constructs of such a language has emerged (e.g., path expressions), but there is no standard language. However, standardized languages for data models such as XML and RDF have emerged (XQuery for XML and SPARQL for RDF). We postpone discussion of these to Sect. 12.6 where we focus on web data integration

12.3.1 Semistructured Data Approach

One way to approach querying the web data is to treat it as a collection of semistructured data. Then, models and languages that have been developed for this purpose can be used to query the data. Semistructured data models and languages were not originally developed to deal with web data; rather they addressed the requirements of growing data collections that did not have as strict a schema as their relational counterparts. However, since these characteristics are also common to web data, later studies explored their applicability in this domain. We demonstrate this approach using a particular model (OEM) and a language (Lorel), but other approaches such as UnQL are similar.

OEM (Object Exchange Model) is a self-describing semistructured data model. Self-describing means that each object specifies the schema that it follows.

An OEM object is defined as a four-tuple $\langle \text{label}, \text{type}, \text{value}, \text{oid} \rangle$, where *label* is a character string describing what the object represents, *type* specifies the type of the object's value, *value* is obvious, and *oid* is the object identifier that distinguishes it from other objects. The type of an object can be *atomic*, in which case the object is called an *atomic object*, or *complex*, in which case the object is called a *complex object*. An atomic object contains a primitive value such as an integer, a real, or a string, while a complex object contains a set of other objects, which can themselves be atomic or complex. The value of a complex object is a set of oids.

Example 12.2 Let us consider a bibliographic database that consists of a number of documents. A snapshot of an OEM representation of such a database is given in Fig. 12.4. Each line shows one OEM object and the indentation is provided to simplify the display of the object structure. For example, the second line

```

<bib, complex, {&o2, &o22, &o34}, &o1>
  <doc, complex, {&o3, &o6, &o7, &o20, &o22}, &o2>
    <authors, complex, {&o4, &o5}, &o3>
      <author, string, "M. Tamer Ozsu", &o4>
      <author, string, "Patrick Valduriez", &o5>
    <title, string, "Principles of Distributed ...", &o6>
    <chapters, complex, {&o8, &o11, &o14, &o9}, &o7>
      <chapter, complex, {&o9, &o10}, &o8>
        <heading, string, "...", &o9>
        <body, string, "...", &o10>
        ...
      <chapter, complex, {&o18, &o19}, &9>
        <heading, string, "...", &o18>
        <body, string, "...", &o19>
    <what, string, "Book", &o20>
    <price, float, 98.50, &o21>
  <doc, complex, {&o23, &o25, &o26, &o27, &o28}, &o22>
    <authors, complex, {&o24, &o4}, &o23>
      <author, string, "Yingying Tao", &o24>
    <title, string, "Mining data streams ...", &o25>
    <venue, string, "CIKM", &o26>
    <year, integer, 2009, &o27>
    <sections, complex, {&o29, &o30, &o31, &o32, &o33}, &28>
      <section, string, "...", &o29>
      ...
    <section, string, "...", &o33>
  <doc, complex, {&o16,&o9,&o7,&o18,&o19,&o20,&o21},&o34>
    <author, string, "Anthony Bonato", &o35>
    <title, string, "A Course on the Web Graph", &o36>
    <what, string, "Book", &o20>
    <ISBN, string, "TK5105.888.B667", &o37>
    <chapters, complex, {&o39, &o42, &o45}, &o38>
      <chapter, complex, {&o40, &o41}, &o39>
        <heading, string, "...", &o40>
        <body, string, "...", &o41>
      <chapter, complex, {&o43, &o44}, &o42>
        <heading, string, "...", &o43>
        <body, string, "...", &o44>
      <chapter, complex, {&o46, &o47}, &45>
        <heading, string, "...", &o46>
        <body, string, "...", &o47>
    <publisher, string, "AMS", &o48>

```

Fig. 12.4 An example OEM specification

`<doc, complex, &o3, &o6, &o7, &o20, &o21, &o2>` defines an object whose label is `doc`, type is `complex`, oid is `&o2`, and whose value consists of objects whose oids are `&o3`, `&o6`, `&o7`, `&o20`, and `&o21`.

This database contains three documents (`&o2`, `&o22`, `&o34`); the first and third are books and the second is an article. There are commonalities among the two books (and even the article), but there are differences as well. For example, `&o2` has

the price information that o_{34} does not have, while o_{34} has ISBN and publisher information that o_{2} does not have. ♦

As noted earlier, OEM data are self-describing, where each object identifies itself through its type and its label. It is easy to see that the OEM data can be represented as a vertex-labeled graph where the vertices correspond to OEM objects and the edges correspond to the subobject relationship. The label of a vertex is the oid and the label of the corresponding object vertex. However, it is quite common in literature to model the data as an edge-labeled graph: if object o_j is a subobject of object o_i , then o_j 's label is assigned to the edge connecting o_i to o_j , and the oids are omitted as vertex labels. In Example 12.3, we use a vertex and edge-labeled representation that shows oids as vertex labels and assigns edge labels as described above.

Example 12.3 Figure 12.5 depicts the vertex and edge-labeled graph representation of the example OEM database given in Example 12.2. Normally, each terminal vertex (i.e., no outgoing edges) also contains the value of that object. To simplify exposition of the idea, we do not show the values. ♦

The semistructured approach fits reasonably well for modeling web data since it can be represented as a graph. Furthermore, it accepts that data may have some structure, but this may not be as rigid, regular, or complete as that of traditional databases. The users do not need to be aware of the complete structure when they query the data. Therefore, expressing a query should not require full knowledge of the structure. These graph representations of data at each data source are generated by wrappers that we discussed in Sect. 7.2.

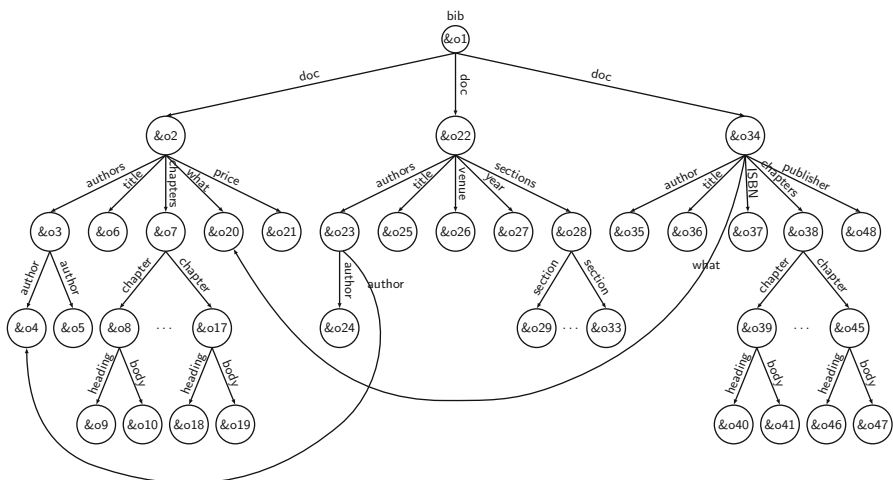


Fig. 12.5 The corresponding OEM graph for the OEM database of Example 12.2

A number of languages have been developed to query semistructured data. As noted above, we will focus our discussion by considering a particular language, Lorel, but other languages are similar in their basic approaches.

Lorel has the familiar `SELECT-FROM-WHERE` structure, but allows path expressions in the `SELECT`, `FROM` and `WHERE` clauses. The fundamental construct in forming Lorel queries is, therefore, a *path expression*. In its simplest form, a path expression in Lorel is a sequence of labels starting with an object name or a variable denoting an object. For example, `bib.doc.title` is a path expression whose interpretation is to start at `bib` and follow the edge-labeled `doc` and then follow the edge-labeled `title`. Note that there are three paths in Fig. 12.5 that would satisfy this expression: (i) `&o1.doc:&o2.title:&o6`, (ii) `&o1.doc:&o22.title:&o25`, and (iii) `&o1.doc:&o34.title:&o36`. Each of these is called a *data path*. In Lorel, path expressions can be more complex regular expressions such that what follows the object name or variable is not only a label, but more general expressions that can be constructed using conjunction, disjunction (`()`), iteration (`?` to mean 0 or 1 occurrences, `+` to mean 1 or more, and `*` to mean 0 or more), and wildcards (`#`).

Example 12.4 The following are examples of acceptable path expressions in Lorel:

- (a) `bib.doc(.authors)?.author` : start from `bib`, follow `doc` edge and the `author` edge with an optional `authors` edge in between.
- (b) `bib.doc.#.author` : start from `bib`, follow `doc` edge, then an arbitrary number of edges with unspecified labels (using the wildcard `#`), and follow the `author` edge.
- (c) `bib.doc.%price` : start from `bib`, follow `doc` edge, then an edge whose label has the string “price” preceded by some characters.



Example 12.5 The following are example Lorel queries that use some of the path expressions given in Example 12.4:

- (a) Find the titles of documents written by Patrick Valduriez.

```
SELECT D.title
FROM bib.doc D
WHERE bib.doc(.authors)?.author =
      "Patrick Valduriez"
```

In this query, the `FROM` clause restricts the scope to documents (`doc`), and the `SELECT` clause specifies the nodes reachable from documents by following the `title` label. We could have specified the `WHERE` predicate as

```
D(.authors)?.author = "Patrick Valduriez".
```


(b) Find the authors of all books whose price is under \$100.

```
SELECT D(.authors)?.author
FROM   bib.doc D
WHERE  D.what = "Books" AND D.price < 100
```



Semistructured data approach to modeling and querying web data is simple and flexible. It also provides a natural way to deal with containment structure of web objects, thereby supporting, to some extent, the link structure of web pages. However, there are also deficiencies of this approach. The data model is too simple—it does not include a record structure (each vertex is a simple entity) nor does it support ordering as there is no imposed ordering among the vertices of an OEM graph. Furthermore, the support for links is also relatively rudimentary, since the model or the languages do not differentiate between different types of links. The links may show either subpart relationships among objects or connections between different entities that correspond to vertices. These cannot be separately modeled, nor can they be easily queried.

Finally, the graph structure can get quite complicated, making it difficult to query. Although Lorel provides a number of features (such as wildcards) to make querying easier, the examples above indicate that a user still needs to know the general structure of the semistructured data. The OEM graphs for large databases can become quite complicated, and it is hard for users to form the path expressions. The issue, then, is how to “summarize” the graph so that there might be a reasonably small schema-like description that might aid querying. For this purpose, a construct called a DataGuide has been proposed. A DataGuide is a graph where each path in the corresponding OEM graph occurs only once. It is dynamic in that as the OEM graph changes, the corresponding DataGuide is updated. Thus, it provides concise and accurate structural summaries of semistructured databases and can be used as a lightweight schema, which is useful for browsing the database structure, formulating queries, storing statistical information, and enabling query optimization.

Example 12.6 The DataGuide corresponding to the OEM graph in Example 12.3 is given in Fig. 12.6. 

12.3.2 Web Query Language Approach

The approaches in this category are aimed to directly address the characteristics of web data, particularly focusing on handling *links* properly. Their starting point is to overcome the shortcomings of keyword search by providing proper abstractions for capturing the content structure of documents (as in semistructured data approaches)

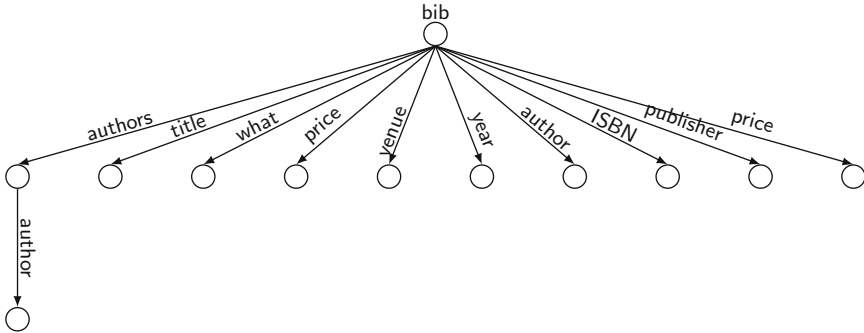


Fig. 12.6 The DataGuide corresponding to the OEM graph of Example 12.3

as well as the external links. They combine the content-based queries (e.g., keyword expressions) and structure-based queries (e.g., path expressions).

A number of languages have been proposed specifically to deal with web data, and these can be categorized as first generation and second generation. The first generation languages model the web as interconnected collection of *atomic* objects. Consequently, these languages can express queries that search the link structure among web objects and their textual content, but they cannot express queries that exploit the document structure of these web objects. The second generation languages model the web as a linked collection of *structured* objects, allowing them to express queries that exploit the document structure similar to semistructured languages. First generation approaches include WebSQL, W3QL, and WebLog, while second generation approaches include WebOQL and StruQL. We will demonstrate the general ideas by considering one first generation language (WebSQL) and one second generation language (WebOQL).

WebSQL is one of the early query languages that combines searching and browsing. It directly addresses web data as captured by web documents (usually in HTML format) that have some content and may include links to other pages or other objects (e.g., PDF files or images). It treats links as first-class objects, and identifies a number of different types of links that we will discuss shortly. As before, the structure can be represented as a graph, but WebSQL captures the information about web objects in two *virtual* relations:

```

DOCUMENT (URL, TITLE, TEXT, TYPE, LENGTH, MODIF)
LINK (BASE, HREF, LABEL)

```

DOCUMENT relation holds information about each web document where URL identifies the web object and is the primary key of the relation, TITLE is the title of the web page, TEXT is its text content of the web page, TYPE is the type of the web object (HTML document, image, etc.), LENGTH is self-explanatory, and MODIF is the last modification date of the object. Except URL, all other attributes can have null values. LINK relation captures the information about links where BASE is the URL

of the HTML document that contains the link, `HREF` is the URL of the document that is referenced, and `LABEL` is the label of the link as defined earlier.

WebSQL defines a query language that consists of SQL plus path expressions. The path expressions are more powerful than their counterparts in Lorel; in particular, they identify different types of links:

- (a) *interior link* that exists within the same document (`#>`)
- (b) *local link* that is between documents on the same server (`->`)
- (c) *global link* that refers to a document on another server (`=>`)
- (d) *null path* (`=`)

These link types form the alphabet of the path expressions. Using them, and the usual constructors of regular expressions, different paths can be specified as in Example 12.7.

Example 12.7 The following are examples of possible path expressions that can be specified in WebSQL.

- (a) `-> | =>`: a path of length one, either local or global
- (b) `->*`: local path of any length
- (c) `=>->*`: as above, but in other servers
- (d) `-> | =>*`: the reachable portion of the web



In addition to path expressions that can appear in queries, WebSQL allows scoping within the **FROM** clause in the following way:

```
FROM Relation SUCH THAT domain-condition
```

where `domain-condition` can be either a path expression, or can specify a text search using **MENTIONS**, or can specify that an attribute (in the **SELECT** clause) is equal to a web object. Of course, following each relation specification, there could be a variable ranging over the relation—this is standard SQL. The following example queries (taken from with minor modifications) demonstrate the features of WebSQL.

Example 12.8 Following are some examples of WebSQL:

- (a) The first example we consider simply searches for all documents about “hypertext” and demonstrates the use of **MENTIONS** to scope the query.

```
SELECT D.URL, D.TITLE
FROM    DOCUMENT D
        SUCH THAT D MENTIONS "hypertext"
WHERE   D.TYPE = "text/html"
```

- (b) The second example demonstrates two scoping methods as well as a search for links. The query is to find all links to applets from documents about “Java.”

```

SELECT A.LABEL, A.HREF
FROM   DOCUMENT D SUCH THAT D MENTIONS "Java"
        ANCHOR A SUCH THAT BASE=X
WHERE  A.LABEL = "applet"

```

- (c) The third example demonstrates the use of different link types. It searches for documents that have the string “database” in their title that are reachable from the ACM Digital Library home page through paths of length two or less containing only local links.

```

SELECT D.URL, D.TITLE
FROM   DOCUMENT D SUCH THAT
        "http://www.acm.org/dl"=| -> | ->-> D
WHERE  D.TITLE CONTAINS "database"

```

- (d) The final example demonstrates the combination of content and structure specifications in a query. It finds all documents mentioning “Computer Science” and all documents that are linked to them through paths of length two or less containing only local links.

```

SELECT D1.URL, D1.TITLE, D2.URL, D2.TITLE
FROM   DOCUMENT D1 SUCH THAT
        D1 MENTIONS "Computer Science",
        DOCUMENT D2 SUCH THAT D1=| -> | ->-> D2

```



WebSQL can query web data based on the links and the textual content of web documents, but it cannot query the documents based on their structure. This limitation is the consequence of its data model that treats the web as a collection of atomic objects.

The second generation languages, such as WebOQL, address this shortcoming by modeling the web as a graph of structured objects. In a way, they combine some features of semistructured data approaches with those of first generation web query models.

WebOQL’s main data structure is a *hypertree*, which is an ordered edge-labeled tree with two types of edges: internal and external. An *internal edge* represents the internal structure of a web document, while an *external edge* represents a reference (i.e., hyperlink) among objects. Each edge is labeled with a record that consists of a number of attributes (fields). An external edge has to have a URL attribute in its record and cannot have descendants (i.e., they are the leaves of the hypertree).

Example 12.9 Let us revisit Example 12.2 and assume that instead of modeling the documents in a bibliography, it models the collection of documents about data management over the web. A possible (partial) hypertree for this example is given

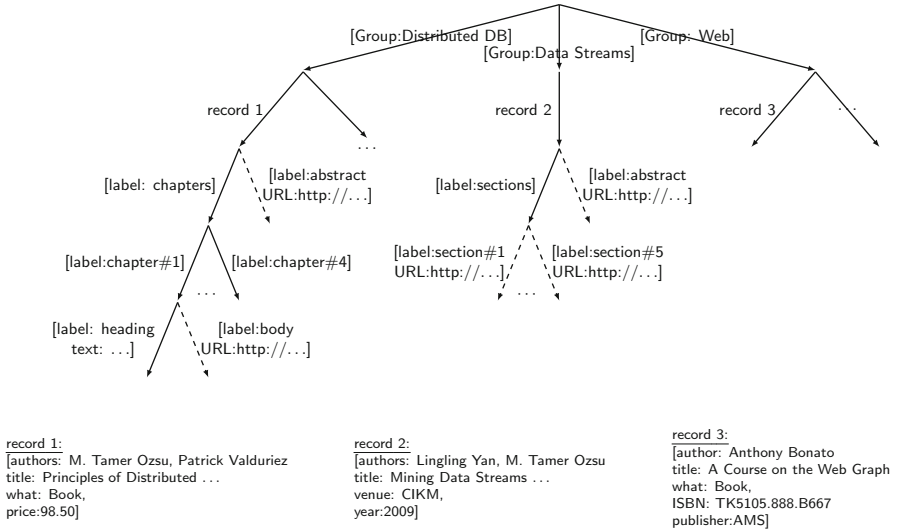


Fig. 12.7 The hypertree example

in Fig. 12.7. Note that we have made one revision to facilitate some of the queries to be discussed later: we added an abstract to each document.

In Fig. 12.7, the documents are first grouped along a number of topics as indicated in the records attached to the edges from the root. In this representation, the internal links are shown as solid edges and external links as dashed edges. Recall that in OEM (Fig. 12.5), the edges represent both attributes (e.g., author) and document structure (e.g., chapter). In the WebOQL model, the attributes are captured in the records that are associated with each edge, while the (internal) edges represent the document structure. ♦

Using this model, WebOQL defines a number of operators over trees:

- Prime:** returns the first subtree of its argument (denoted \prime).
- Peek:** extracts a field from the record that labels the first outgoing edges of its document. For example, if x points to the root of the subtree reached from the “Groups = Distributed DB” edge, x .authors would retrieve “M. Tamer Ozs, Patrick Valduriez.”
- Hang:** builds an edge-labeled tree with a record formed with the arguments (denoted as $[]$).

Example 12.10 Let us assume that the tree depicted in Fig. 12.8a is retrieved as a result of a query (call it Q1). Then the expression [\prime Label: “Papers by Ozs” / Q1] results in the tree depicted in Fig. 12.8b. ♦

Concatenate: combines two trees (denoted $+$).

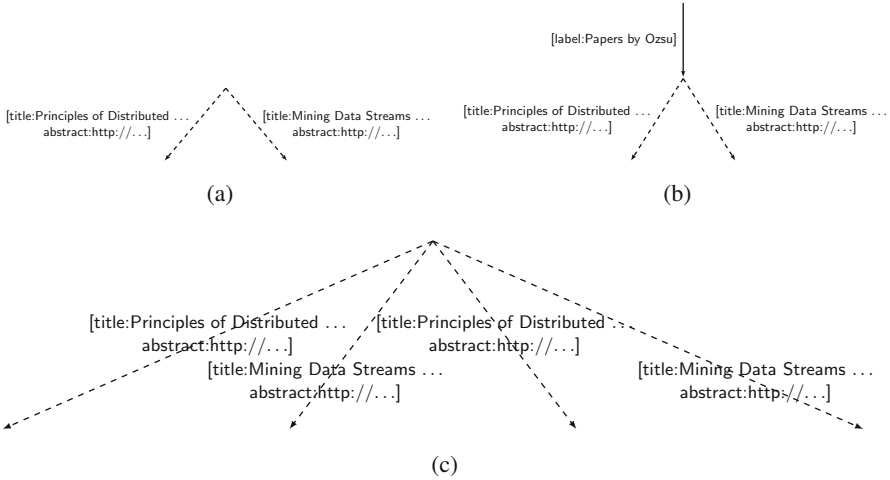


Fig. 12.8 Examples of Hang and Concatenate operators

Example 12.11 Again, assuming that the tree depicted in Fig. 12.8a is retrieved as a result of query Q1, Q1+Q2 produces tree in Fig. 12.8c. ♦

Head: returns the first simple tree of a tree (denoted &). A simple tree of a tree t are the trees composed of one edge followed by a (possibly null) tree that originates from t 's root.

Tail: returns all but the first simple tree of a tree (denoted !).

In addition to these, WebOQL introduces a string pattern matching operator (denoted \sim) whose left argument is a string and right argument is a string pattern. Since the only data type supported by the language is string, this is an important operator.

WebOQL is a functional language, so complex queries can be composed by combining these operators. In addition, it allows these operators to be embedded in the usual SQL (or OQL) style queries as demonstrated by the following example.

Example 12.12 Let `dbDocuments` denote the documents in the database shown in Fig. 12.7. Then the following query finds the titles and abstracts of all documents authored by “Ozsu” producing the result depicted in Fig. 12.8a.

```

SELECT y.title, y'.URL
FROM x IN dbDocuments, y IN x'
WHERE y.authors  $\sim$  "Ozsu"
    
```

The semantics of this query is as follows: The variable x ranges over the simple trees of `dbDocuments`, and, for a given x value, y iterates over the simple trees of the single subtree of x . It peeks into the record of the edge and if the `authors` value matches “Ozsu” (using the string matching operator \sim), then it constructs a

tree whose label is the `title` attribute of the record that `y` points to and the `URL` attribute value of the subtree. ♦

The web query languages discussed in this section adopt a more powerful data model than the semistructured approaches. The model can capture both the document structure and the connectedness of web documents. The languages can then exploit these different edge semantics. Furthermore, as we have seen from the WebOQL examples, the queries can construct new structures as a result. However, formation of these queries still requires some knowledge about the graph structure.

12.4 Question Answering Systems

In this section, we discuss an interesting and unusual (from a database perspective) approach to accessing web data: question answering (QA) systems. These systems accept natural language questions that are then analyzed to determine the specific query that is being posed. They then conduct a search to find the appropriate answer.

Question answering systems have grown within the context of IR systems where the objective is to determine the answer to posed queries within a well-defined corpus of documents. These are usually referred to as *closed domain* systems. They extend the capabilities of keyword search queries in two fundamental ways. First, they allow users to specify complex queries in natural language that may be difficult to specify as simple keyword search requests. In the context of web querying, they also enable asking questions without a full knowledge of the data organization. Sophisticated natural language processing (NLP) techniques are then applied to these queries to understand the specific query. Second, they search the corpus of documents and return explicit answers rather than links to documents that may be relevant to the query. This does not mean that they return exact answers as traditional DBMSs do, but they may return a (ranked) list of explicit responses to the query, rather than a set of web pages. For example, a keyword search for “President of USA” using a search engine would return the (partial) result in Fig. 12.9. The user is expected to find the answer within the pages whose URLs and short descriptions (called snippets) are included on this page (and several more). On the other hand, a similar search using a natural language question “Who is the president of USA?” might return a ranked list of presidents’ names (the exact type of answer differs among different systems).

Question answering systems have been extended to operate on the web. In these systems, the web is used as the corpus (hence they are called *open domain* systems). The web data sources are accessed using wrappers that are developed for them to obtain answers to questions. A number of question answering systems have been developed with different objectives and functionalities, such as Mulder, WebQA, Start, and Tritus. There are also commercial systems with varying capabilities (e.g., Wolfram Alpha <http://www.wolframalpha.com/>).

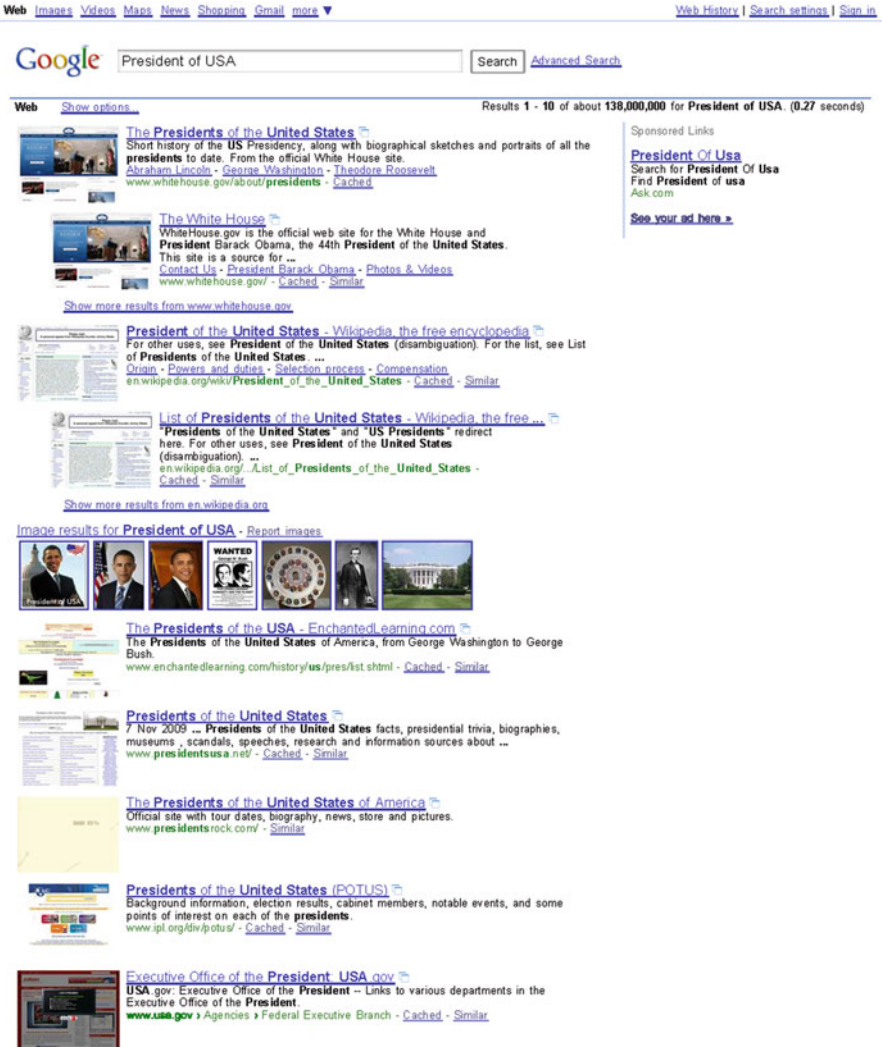


Fig. 12.9 Keyword search example

We describe the general functionality of these systems using the reference architecture given in Fig. 12.10. Preprocessing, which is not employed in all systems, is an offline process to extract and enhance the rules that are used by the systems. In many cases, these are analyses of documents extracted from the web or returned as answers to previously asked questions in order to determine the most effective query structures into which a user question can be transformed. These transformation rules are stored in order to use them at runtime while answering the user questions. For example, Tritus employs a learning-based approach that uses

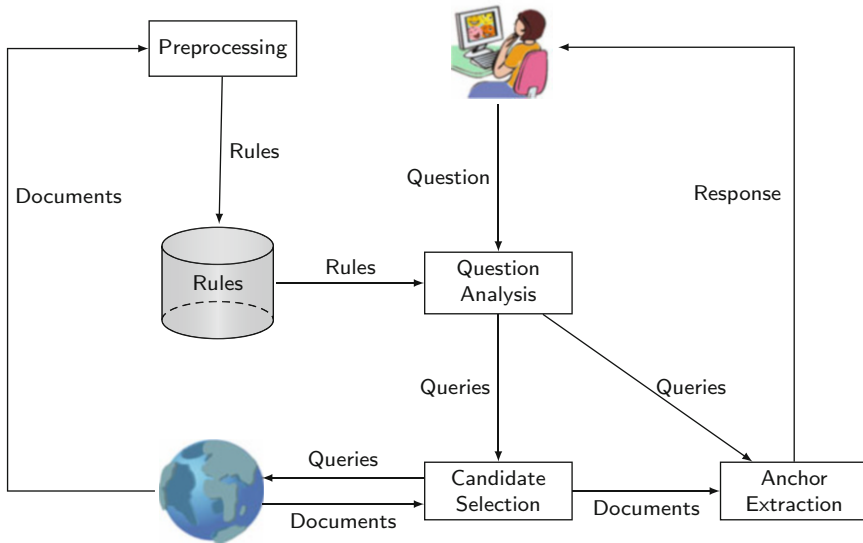


Fig. 12.10 General architecture of QA systems

a collection of frequently asked questions and their correct answers as a training dataset. In a three-stage process, it attempts to guess the structure of the answer by analyzing the question and searching for the answer in the collection. In the first stage, the question is analyzed to extract the *question phrase* (e.g., in the question “What is a hard disk?,” “What is a” is a question phrase). This is used to classify the question. In the second phase, it analyzes the question-answer pairs in the training data and generates *candidate transforms* for each question phrase (e.g., for the question phrase “What is a,” it generates “refers to,” “stands for,” etc.). In the third stage, each candidate transform is applied to the questions in the training dataset, and the resulting transformed queries are sent to different search engines. The similarities of the returned answers with the actual answers in the training data are calculated, and, based on these, a ranking is done for candidate transforms. The ranked transformation rules are stored for later use during runtime execution of questions.

The natural language question that is posed by a user first goes through the question analysis process. The objective is to understand the question issued by the user. Most of the systems try to guess the type of the answer in order to categorize the question, which is used in translating the question into queries and also in answer extraction. If preprocessing has been done, the transformation rules that have been generated are used to assist the process. Although the general goals are the same, the approaches used by different systems vary considerably depending on the sophistication of the NLP techniques employed by the systems (this phase is usually all about NLP). For example, question analysis in Mulder incorporates three phases: question parsing, question classification, and query generation. Query

parsing generates a parse tree that is used in query generation and in answer extraction. Question classification, as its name implies, categorizes the question in one of a number of classes: e.g., *nominal* is for nouns, *numerical* is for numbers, and *temporal* is for dates. This type of categorization is done in most of the QA systems because it eases the answer extraction. Finally, query generation phase uses the previously generated parse tree to construct one or more queries that can be executed to obtain the answers to the question. Mulder uses four different methods in this phase.

- Verb conversion: Auxiliary and main verb is replaced by the conjugated verb (e.g., “When did Nixon visit China?” is converted to “Nixon visited China”).
- Query expansion: Adjective in the question phrase is replaced by its attribute noun (e.g., “How tall is Mt. Everest?” is converted to “The height of Everest is”).
- Noun phrase formation: Some noun phrases are quoted in order to give them together to the search engine in the next stage.
- Transformation: Structure of the question is transformed into the structure of the expected answer type (“Who was the first American in space?” is converted to “The first American in space was”).

Mulder is an example of a system that uses a sophisticated NLP approach to question analysis. At the other end of the spectrum is WebQA, which follows a lightweight approach in question parsing.

Once the question is analyzed and one or more queries are generated, the next step is to generate candidate answers. The queries that were generated at question analysis stage are used at this step to perform keyword search for relevant documents. Many of the systems simply use the general-purpose search engines in this step, while others also consider additional data sources that are available on the web. For example, CIA’s World Factbook (<https://www.cia.gov/library/publications/the-world-factbook/>) is a very popular source for reliable factual data about countries. Similarly, weather information may be obtained very reliably from a number of weather data sources such as the Weather Network (<http://www.theweathernetwork.com/>) or Weather Underground (<http://www.wunderground.com/>). These additional data sources may provide better answers in some cases and different systems take advantage of these to differing degrees. Since different queries can be better answered by different data sources (and, sometimes, even by different search engines), an important aspect of this processing stage is the choice of the appropriate search engine(s)/data source(s) to consult for a given query. The naive alternative of submitting the queries to all search engines and data sources is not a wise decision, since these operations are quite costly over the web. Usually, the category information is used to assist the choice of the appropriate sources, along with a ranked listing of sources and engines for different categories. For each search engine and data source, wrappers need to be written to convert the query into the format of that data source/search engine and convert the returned result documents into a common format for further analysis.

In response to queries, search engines return links to the documents together with short snippets, while other data sources return results in a variety of formats.

The returned results are normalized into “records.” The direct answers need to be extracted from these records, which is the function of the answer extraction phase. Various text processing techniques can be used to match the keywords to (possibly parts of) the returned records. Subsequently, these results need to be ranked using various information retrieval techniques (e.g., word frequencies, inverse document frequency). In this process, the category information that is generated during question analysis is used. Different systems employ different notions of the appropriate answer. Some return a ranked list of direct answers (e.g., if the question is “Who invented the telephone,” they would return “Alexander Graham Bell” or “Graham Bell” or “Bell,” or all of them in ranked order⁴), while others return a ranked order of the portion of the records that contain the keywords in the query (i.e., a summary of the relevant portion of the document).

Question answering systems are very different than the other web querying approaches we have discussed in previous sections. They are more flexible in what they offer users in terms of querying without any knowledge of the organization of web data. On the other hand, they are constrained by idiosyncrocies of natural language, and the difficulties of natural language processing.

12.5 Searching and Querying the Hidden Web

Currently, most general-purpose search engines only operate on the PIW while a considerable amount of the valuable data are kept in hidden databases, either as relational data, as embedded documents, or in many other forms. The current trend in web search is to find ways to search the hidden web as well as the PIW, for two main reasons. First is the size—the size of the hidden web (in terms of generated HTML pages) is considerably larger than the PIW, therefore the probability of finding answers to users’ queries is much higher if the hidden web can also be searched. The second is in data quality—the data stored in the hidden web are usually of much higher quality than those found on public web pages since they are properly curated. If they can be accessed, the quality of answers can be improved.

However, searching the hidden web faces many challenges, the most important of which are the following:

1. Ordinary crawlers cannot be used to search the hidden web, since there are neither HTML pages, nor hyperlinks to crawl.
2. Usually, the data in hidden databases can be only accessed through a search interface or a special interface, requiring access to this interface.
3. In most (if not all) cases, the underlying structure of the database is unknown, and the data providers are usually reluctant to provide any information about their data that might help in the search process (possibly due to the overhead

⁴The inventor of the telephone is a subject of controversy, with multiple claims to the invention. We’ll go with Bell in this example since he was the first one to patent the device.

of collecting this information and maintaining it). One has to work through the interfaces provided by these data sources.

In the remainder of this section, we discuss these issues as well as some proposed solutions.

12.5.1 Crawling the Hidden Web

One approach to address the issue of searching the hidden web is to try crawling in a manner similar to that of the PIW. As already mentioned, the only way to deal with hidden web databases is through their search interfaces. A hidden web crawler should be able to perform two tasks: (a) submit queries to the search interface of the database, and (b) analyze the returned result pages and extract relevant information from them.

12.5.1.1 Querying the Search Interface

One approach is to analyze the search interface of the database, and build an internal representation for it. This internal representation specifies the fields used in the interface, their types (e.g., text boxes, lists, checkboxes, etc.), their domains (e.g., specific values as in lists, or just free text strings as in text boxes), and also the labels associated with these fields. Extracting these labels requires an exhaustive analysis of the HTML structure of the page.

Next, this representation is matched with the system's task-specific database. The matching is based on the labels of the fields. When a label is matched, the field is then populated with the available values for this field. The process is repeated for all possible values of all fields in the search form, and the form is submitted with every combination of values and the results are retrieved.

Another approach is to use agent technology. In this case, *hidden web agents* are developed that interact with the search forms and retrieve the result pages. This involves three steps: (a) finding the forms, (b) learning to fill the forms, and (c) identifying and fetching the target (result) pages.

The first step is accomplished by starting from a URL (an entry point), traversing links, and using some heuristics to identify HTML pages that contain forms, excluding those that contain password fields (e.g., login, registration, purchase pages). The form filling task depends on identifying labels and associating them with form fields. This is achieved using some heuristics about the location of the label relative to the field (on the left or above it). Given the identified labels, the agent determines the application domain that the form belongs to, and fills the fields with values from that domain in accordance with the labels (the values are stored in a repository accessible to the agent).

12.5.1.2 Analyzing the Result Pages

Once the form is submitted, the returned page has to be analyzed, for example, to see if it is a data page or a search-refining page. This can be achieved by matching values in this page with values in the agent's repository. Once a data page is found, it is traversed, as well as all pages that it links to (especially pages that have more results), until no more pages can be found that belong to the same domain.

However, the returned pages usually contain a lot of irrelevant data, in addition to the actual results, since most of the result pages follow some template that has a considerable amount of text used only for presentation purposes. A method to identify web page templates is to analyze the textual contents and the adjacent tag structures of a document in order to extract query-related data. A web page is represented as a sequence of text segments, where a text segment is a piece of tag encapsulated between two tags. The mechanism to detect templates is as follows:

1. Text segments of documents are analyzed based on textual contents and their adjacent tag segments.
2. An initial template is identified by examining the first two sample documents.
3. The template is then generated if matched text segments along with their adjacent tag segments are found from both documents.
4. Subsequent retrieved documents are compared with the generated template. Text segments that are not found in the template are extracted for each document to be further processed.
5. When no matches are found from the existing template, document contents are extracted for the generation of future templates.

12.5.2 Metasearching

Metasearching is another approach for querying the hidden web. Given a user query, a metasearcher performs the following tasks:

1. Database selection: selecting the databases(s) that are most relevant to the user's query. This requires collecting some information about each database. This information is known as a *content summary*, which is statistical information, usually including the *document frequencies* of the words that appear in the database.
2. Query translation: translating the query to a suitable form for each database (e.g., by filling certain fields in the database's search interface).
3. Result merging: collecting the results from the various databases, merging them (and most probably, ordering them), and returning them to the user.

We discuss the important phases of metasearching in more detail below.

12.5.2.1 Content Summary Extraction

The first step in metasearching is to compute content summaries. In most of the cases, the data providers are not willing to go through the trouble of providing this information. Therefore, the metasearcher itself extracts this information.

A possible approach is to extract a document sample set from a given database D and compute the frequency of each observed word w in the sample, $SampleDF(w)$. The technique works as follows:

1. Start with an empty content summary where $SampleDF(w) = 0$ for each word w , and a general (i.e., not specific to D), comprehensive word dictionary.
2. Pick a word and send it as a query to database D .
3. Retrieve the top- k documents from among the returned documents.
4. If the number of retrieved documents exceeds a prespecified threshold, stop. Otherwise continue the sampling process by returning to Step 2.

There are two main versions of this algorithm that differ in how Step 2 is executed. One of the algorithms picks a random word from the dictionary. The second algorithm selects the next query from among the words that have been already discovered during sampling. The first constructs better profiles, but is more expensive.

An alternative is to use a focused probing technique that can actually classify the databases into a hierarchical categorization. The idea is to preclassify a set of training documents into some categories, and then extract different terms from these documents and use them as query probes for the database. The single-word probes are used to determine the *actual* document frequencies of these words, while only *sample* document frequencies are computed for other words that appear in longer probes. These are used to estimate the actual document frequencies for these words.

Yet another approach is to start by randomly selecting a term from the search interface itself, assuming that, most probably, this term will be related to the contents of the database. The database is queried for this term, and the top- k documents are retrieved. A subsequent term is then randomly selected from terms extracted from the retrieved documents. The process is repeated until a predefined number of documents are retrieved, and then statistics are calculated based on the retrieved documents.

12.5.2.2 Database Categorization

A good approach that can help the database selection process is to categorize the databases into several categories (for example, as Yahoo directory). Categorization facilitates locating a database given a user's query, and makes most of the returned results relevant to the query.

If the focused probing technique is used for generating content summaries, then the same algorithm can probe each database with queries from some category and

count the number of matches. If the number of matches exceeds a certain threshold, the database is said to belong to this category.

Database Selection

Database selection is a crucial task in the metasearching process, since it has a critical impact on the efficiency and effectiveness of query processing over multiple databases. A database selection algorithm attempts to find the best set of databases, based on information about the database contents, on which a given query should be executed. Usually, this information includes the number of different documents that contain each word (known as the document frequency), as well as some other simple related statistics, such as the number of documents stored in the database. Given these summaries, a database selection algorithm estimates how relevant each database is for a given query (e.g., in terms of the number of matches that each database is expected to produce for the query).

GLOSS is a simple database selection algorithm that assumes that query words are independently distributed over database documents to estimate the number of documents that match a given query. GLOSS is an example of a large family of database selection algorithms that rely on content summaries. Furthermore, database selection algorithms expect such content summaries to be accurate and up-to-date.

The focused probing algorithm discussed above exploits the database categorization and content summaries for database selection. This algorithm consists of two basic steps: (1) propagate the database content summaries to the categories of the hierarchical classification scheme, and (2) use the content summaries of categories and databases to perform database selection hierarchically by zooming in on the most relevant portions of the topic hierarchy. This results in more relevant answers to the user's query since they only come from databases that belong to the same category as the query itself.

Once the relevant databases are selected, each database is queried, and the returned results are merged and sent back to the user.

12.6 Web Data Integration

In Chap. 7 we discussed the integration of databases, each of which have well-defined schemas. The techniques discussed in that chapter are mostly appropriate when enterprise data is considered. When we wish to provide integrated access to web data sources, the problem becomes more complex—all the characteristics of “big data” play a role. In particular, the data may not have a proper schema, and if it does, the data sources are so varied that the schemas are widely different, making schema matching a real challenge. In addition, the amount of data and even the number of data sources are significantly higher than in an enterprise environment, making manual curation all but impossible. The quality of the data on the web is also

far more suspect than the enterprise data collections that we considered previously, and this increases the importance of data cleaning solutions.

An appropriate approach to web data integration is what is called *pay-as-you-go integration* where the up-front investment to integrate data is significantly reduced, eliminating some of the stages discussed in Chap. 7. Instead, a framework and basic infrastructure is provided for data owners to easily integrate their datasets into a federation. One proposal for the pay-as-you-go approach to web data integration is *data spaces*, which advocates that there should be lightweight integration platform with perhaps rudimentary access opportunities (e.g., keyword search) to start with, and ways to improve the value of the integration over time by providing the opportunity to develop tools for more sophisticated use. Perhaps the *data lakes* that have started to receive attention and that we discussed in Chap. 10, are more advanced versions of the data space proposal.

In this section, we cover some of the approaches that have been developed to address these challenges. In particular, we will look at web tables and fusion tables (Sect. 12.6.1) as a low-overhead integration approach for tabular structured data. We then look at the semantic web and the Linked Open Data (LOD) approach to web data integration (Sect. 12.6.2.3). Finally, we discuss the issues of data cleaning and the use of machine learning techniques in data integration and cleaning at web-scale integration in Sect. 12.6.3.

12.6.1 Web Tables/Fusion Tables

Two popular approaches to lightweight web data integration are *web portals* and *mashups* that aggregate web and other data on specific topics such as travel, hotel bookings, etc. The two differ in the technologies that they use, but that is not important for our discussion. These are examples of “vertically integrated” systems where each mashup or portal targets one domain.

A first question that comes up in developing a mashup is how to find the relevant web data. Web tables project is an early attempt at finding data on the web that has relational table structure and provide access over these tables (the so-called “database-like” tables). The focus is on the open web, and tables in the deep web are not considered as their discovery is a much more difficult problem. Even finding the database-like tables in the open web is not easy since the usual relational table structures (i.e., attribute names) may not exist. Web tables employ a classifier that can group HTML tables as relational and nonrelational. It then provides tools to extract a schema and maintain statistics about the schemas that can be used in search over these tables. Join opportunities across tables are introduced to allow more sophisticated navigation across the discovered tables. Web tables can be viewed as a method to retrieve and query web data, but they also serve as a virtual integration framework for web data with global schema information.

Fusion tables project at Google takes web tables a step further by allowing users to upload their own tables (in various formats) in addition to the discovered

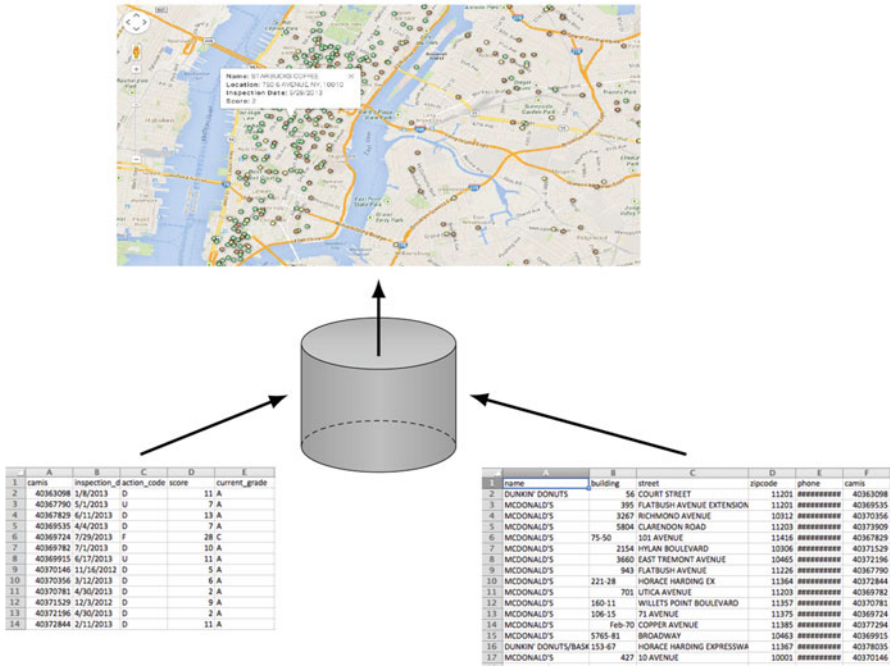


Fig. 12.11 Web tables/fusion tables example

web tables. The fusion table's infrastructure can automatically discover the join attribute across tables and produce integration opportunities. An example is given in Fig. 12.11 which depicts two datasets contributed by two different owners, one on eateries and the other about the scores and grades given to these eateries as a result of inspection. The system would determine that the two datasets can be joined over a common attribute and provide integrated access. Although in this case both tables were contributed by users, in other cases one or both of the tables can be discovered from the web by using the techniques developed by web tables project.

12.6.2 Semantic Web and Linked Open Data

A fundamental contribution of the web is to produce a repository of machine processable data. Semantic web aims to convert this data into machine understandable form by integrating both structured and unstructured data on the web and marking it up semantically. The original semantic web vision includes three components:

- Markup web data so that metadata is captured as annotations;
- Use ontologies for making different data collections understandable; and
- Use logic-based technologies to access both the metadata and the ontologies.

Linked Open Data (LOD) was introduced in 2006 as a clarification of this vision emphasizing the linkages among the data that is part of the semantic web. It set out guidelines for how data should be published on the web to achieve the vision of the semantic web. Thus, the semantic web is a web data integration vision realized through LOD. LOD requirements for publishing (and hence integrating) data on the web are based on four principles:

- All web resources (data) are locally identified by their URIs that serve as names;
- These names are accessible by HTTP;
- Information about web resources/entities are encoded as RDF (Resource Description Framework) triples. In other words, RDF is the semantic web data model (and we discuss it below);
- Connections among datasets are established by data links and publishers of datasets should establish these links so that more data is discoverable.

The LOD, therefore, generates a graph where the vertices are web resources and the edges are the relationships. A simplified form of “LOD graph” as of 2018 is shown in Fig. 12.12 where each vertex represents a dataset (not a web resource) categorized according to by color (e.g., publications, life sciences, social networking) and the size of each vertex represents its in-degree. At that time, LOD consisted of 1,234 datasets with 16,136 links.⁵ We will come back to LOD and the LOD graph shortly.

Semantic web consists of a number of technologies that build upon each other (Fig. 12.13). At the bottom layer, XML provides the language for writing structured web documents and exchanging them easily. On top of this is the RDF that, as noted above, establishes the data model. Although it is not necessary, if a schema over this data is specified, the RDF Schema provides the necessary primitives. Ontologies extend RDF schema with more powerful constructs to specify the relationships among web data. Finally, logic-based declarative rule languages allow applications to define their own rules.

In the remainder we discuss the technologies in the lower layers as these are the minimal requirements.

12.6.2.1 XML

The predominant encoding for web documents has been HTML (which stands for HyperText Markup Language). A web document encoded in HTML consists of *HTML elements* encapsulated by tags as discussed in Sect. 12.6.1 where we also presented approaches to discover structured data in HTML-encoded web documents and integrating them. As noted above, within the context of semantic web, the preferred representation for encoding and exchanging web documents is XML

⁵Statistics obtained from <https://lod-cloud.net>, which should be consulted for up-to-date statistics.

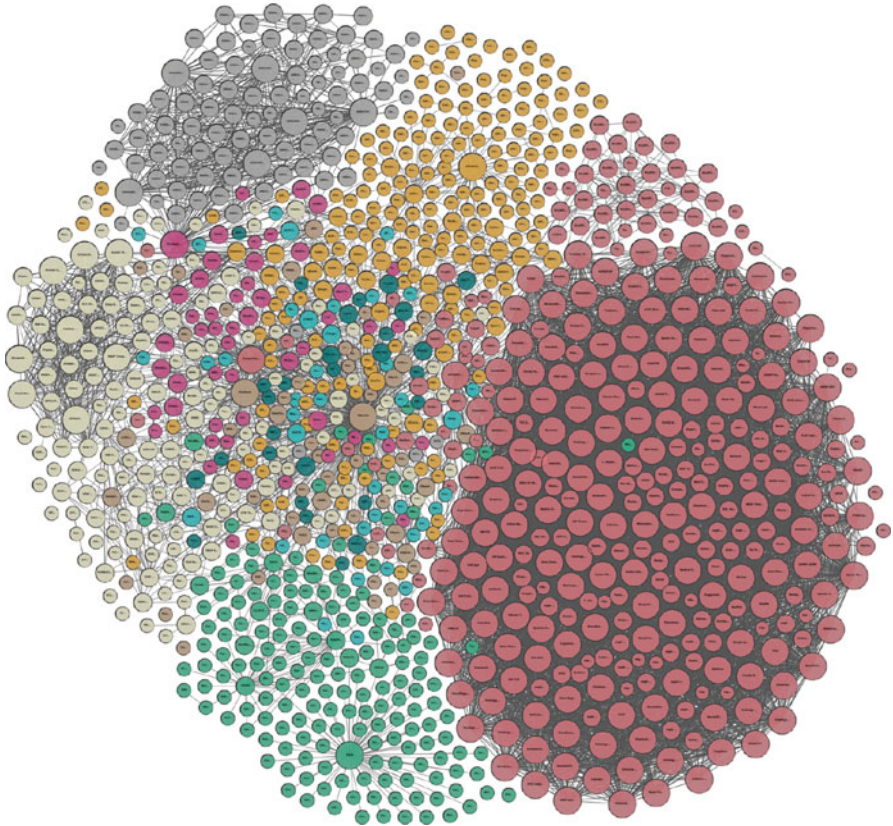


Fig. 12.12 LOD graph as of 2018

Declarative Rule Languages
Ontology Languages
RDF Schema
RDF
XML

Fig. 12.13 Semantic web technologies. Simplified from [Antoniou and Plexousakis 2018]

(which stands for Extensive Markup Language) proposed by the World Wide Web Consortium (W3C).

XML tags (also called markups) divide data into pieces called *elements*, with the objective to provide more semantics to the data. Elements can be nested but they cannot be overlapped. Nesting of elements represents hierarchical relationships between them. As an example, Fig. 12.14 is the XML representation, with slight revisions, of the bibliography data that we had given earlier.

An XML document can be represented as a tree that contains a *root element*, which has zero or more nested subelements (or *child elements*), which can recursively contain subelements. For each element, there are zero or more *attributes* with atomic values assigned to them. An element also contains an optional value. Due to the textual representation of the tree, a total order, called *document order*, is defined on all elements corresponding to the order in which the first character of the elements occurs in the document.

For instance, the root element in Fig. 12.4 is `bib`, which has three child elements: two `book` and one `article`. The first `book` element has an attribute `year` with atomic value "1999", and also contains subelements (e.g., the `title` element). An element can contain a value (e.g., "Principles of Distributed Database Systems" for the element `title`).

Standard XML document definition is a bit more complicated: it can contain ID-IDsREFs, which define references between elements in the same document or in another document. In that case, the document representation becomes a graph. However, it is quite common to use the simpler tree representation, and we'll assume the same in this section and we define it more precisely below.⁶

An XML document is modeled as an ordered, node-labeled tree $T = (V, E)$, where each node $v \in V$ corresponds to an element or attribute and is characterized by:

- a unique identifier denoted by $ID(v)$;
- a unique *kind* property, denoted as $kind(v)$, assigned from the set $\{element, attribute, text\}$;
- a label, denoted by $label(v)$, assigned from some alphabet Σ ;
- a content, denoted by $content(v)$, which is empty for nonleaf nodes and is a string for leaf nodes.

A directed edge $e = (u, v)$ is included in E if and only if:

- $kind(u) = kind(v) = element$, and v is a subelement of u ; or
- $kind(u) = element \wedge kind(v) = attribute$, and v is an attribute of u .

Now that an XML document tree is properly defined, we can define an instance of XML data model as an ordered collection (sequence) of XML document tree nodes or atomic values. A schema may or may not be defined for an XML document, since it is a self-describing format. If a schema is defined for a collection of

⁶In addition, we omit the comment nodes, namespace nodes, and PI nodes from the model.

```

<bib>
<book year = "1999">
<author> M. Tamer Ozsu </author>
<author> Patrick Valduriez </author>
<title> Principles of Distributed ... </title>
<chapters>
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
...
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
</chapters>
<price currency= "USD"> 98.50 </price>
</book>
<article year = "2009">
<author> M. Tamer Ozsu </author>
<author> Yingying Tao </author>
<title> Mining data streams ... </title>
<venue> "CIKM" </venue>
<sections>
<section> ... </section>
...
<section> ... </section>
</sections>
</article>
<book>
<author> Anthony Bonato </author>
<title> A Course on the Web Graph </title>
<ISBN> TK5105.888.B667 </ISBN>
<chapters>
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
<chapter>
<heading> ... </heading>
<body> ... </body>
</chapter>
</chapters>
<publisher> AMS </publisher>
</book>
</bib>

```

Fig. 12.14 An example XML document

XML documents, then each document in this collection conforms to that schema; however, the schema allows for variations in each document, since not all elements or attributes may exist in each document. XML schemas can be defined either using the Document Type Definition (DTD) or XMLSchema. In this section, we will use a simpler schema definition that exploits the graph structure of XML documents as defined above.

An XML *schema graph* is defined as a 5-tuple $\langle \Sigma, \Psi, s, m, \rho \rangle$ where Σ is an alphabet of XML document node types, ρ is the root node type, $\Psi \subseteq \Sigma \times \Sigma$ is a set of edges between node types, $s : \Psi \rightarrow \{\text{ONCE}, \text{OPT}, \text{MULT}\}$ and $m : \Sigma \rightarrow \{\text{string}\}$. The semantics of this definition are as follows: An edge $\psi = (\sigma_1, \sigma_2) \in \Psi$ denotes that an item of type σ_1 may contain an item of type σ_2 . $s(\psi)$ denotes the cardinality of the containment represented by this edge: If $s(\psi) = \text{ONCE}$, then an item of type σ_1 must contain exactly one item of type σ_2 . If $s(\psi) = \text{OPT}$, then an item of type σ_1 may or may not contain an item of type σ_2 . If $s(\psi) = \text{MULT}$, then an item of type σ_1 may contain multiple items of type σ_2 . $m(\sigma)$ denotes the domain of the text content of an item of type σ , represented as the set of all strings that may occur inside such an item.

Using the definition of XML data model and instances of this data model, it is now possible to define the query languages. Expressions in XML query languages take an instance of XML data as input and produce an instance of XML data as output. XPath and XQuery are two query languages proposed by the W3C. Path expressions, that we introduced earlier, are present in both query languages and are arguably the most natural way to query the hierarchical XML data. XQuery defines for more powerful constructs. Although XQuery was the subject of intense research and development efforts in the 2000s, it is not widely used any longer. It is complicated, hard to formulate by users and difficult to optimize by systems. JSON has replaced both XML and XQuery for many applications, as we discussed in Chap. 11, although XML representation remains important for the semantic web (but not XQuery).

12.6.2.2 RDF

RDF is the data model on top of XML and forms a fundamental building block of the semantic web (Fig. 12.13). Although it was originally proposed by W3C as a component of the semantic web, its use is now wider. For example, Yago and DBpedia extract facts from Wikipedia automatically and store them in RDF format to support structural queries over Wikipedia; biologists encode their experiments and results using RDF to communicate among themselves leading to RDF data collections, such as Bio2RDF (bio2rdf.org) and Uniprot RDF (dev.isb-sib.ch/projects/uniprot-rdf). Related to semantic web, LOD project builds a RDF data cloud by linking a large number of datasets, as noted earlier.

RDF models each “fact” as a set of triples (subject, property (or predicate), object), denoted as $\langle s, p, o \rangle$, where *subject* is an entity, class or blank node, a

*property*⁷ denotes one attribute associated with one entity, and *object* is an entity, a class, a blank node, or a literal value. According to the RDF standard, an entity is denoted by a URI (Uniform Resource Identifier) that refers to a named *resource* in the environment that is being modeled. Blank nodes, by contrast, refer to anonymous resources that do not have a name.⁸ Thus, each triple represents a named relationship; those involving blank nodes simply indicate that “something with the given relationship exists, without naming it.”

It is appropriate at this point to briefly talk about the next layer in the semantic web technology stack (Fig. 12.13), namely the RDF Schema (RDFS). It is possible to annotate RDF data with semantic metadata using RDFS, which is also a W3C standard.⁹ This annotation primarily enables reasoning over the RDF data (called *entailment*), and also impacts data organization in some cases, and the metadata can be used for semantic query optimization. We illustrate the fundamental concepts by simple examples using RDFS, which allows the definition of *classes* and *class hierarchies*. RDFS has built-in class definitions—the more important ones being `rdfs:Class` and `rdfs:subClassOf` that are used to define a class and a subclass, respectively (another one, `rdfs:label` is used in our query examples below). To specify that an individual resource is an element of the class, a special property, `rdf:type` is used.

Example 12.13 For example, if we wanted to define a class called `Movies` and two subclasses `ActionMovies` and `Dramas`, this would be accomplished in the following way:

```
Movies rdf:type rdfs:Class .
ActionMovies rdfs:subClassOf Movies .
Dramas rdfs:subClassOf Movies .
```



Formally, a RDF dataset can be defined as follows. Let \mathcal{U} , \mathcal{B} , and \mathcal{L} , denote the sets of all URIs, blank nodes, and literals, respectively. A tuple $(s, p, o) \in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is an *RDF triple*. A set of RDF triples form a *RDF dataset*.

Example 12.14 An example RDF dataset is shown in Fig. 12.15 where the data comes from a number of sources as defined by the URI prefixes.



RDF data can be modeled as an RDF graph as follows. A *RDF graph* is a six-tuple $G = \langle V, L_V, f_V, E, L_E, f_E \rangle$, where

⁷In literature, the terms “property” and “predicate” are used interchangeably; in this paper, we will use “property” consistently.

⁸In much of the research, blank nodes are ignored. Unless explicitly stated otherwise, we will ignore them in this paper as well.

⁹The same annotation can also be done using the ontology languages such as OWL (also a W3C standard) but we will not discuss that topic further.

Prefixes:

`mdb`=<http://data.linkedmdb.org/resource/geo>=<http://sws.geonames.org/>

`bm`=<http://wifo5-03.informatik.uni-mannheim.de/bookmashup/>

`exvo`=<http://lexvo.org/id/>

`wp`=<http://en.wikipedia.org/wiki/>

Subject	Property	Object
<code>mdb: film/2014</code>	<code>rdfs:label</code>	"The Shining"
<code>mdb:film/2014</code>	<code>movie:initial_release_date</code>	"1980-05-23"
<code>mdb:film/2014</code>	<code>movie:director</code>	<code>mdb:director/8476</code>
<code>mdb:film/2014</code>	<code>movie:actor</code>	<code>mdb:actor/29704</code>
<code>mdb:film/2014</code>	<code>movie:actor</code>	<code>mdb: actor/30013</code>
<code>mdb:film/2014</code>	<code>movie:music_contributor</code>	<code>mdb: music_contributor/4110</code>
<code>mdb:film/2014</code>	<code>foaf:based_near</code>	<code>geo:2635167</code>
<code>mdb:film/2014</code>	<code>movie:relatedBook</code>	<code>bm:0743424425</code>
<code>mdb:film/2014</code>	<code>movie:language</code>	<code>lexvo:iso639-3/eng</code>
<code>mdb:director/8476</code>	<code>movie:director_name</code>	"Stanley Kubrick"
<code>mdb:film/2685</code>	<code>movie:director</code>	<code>mdb:director/8476</code>
<code>mdb:film/2685</code>	<code>rdfs:label</code>	"A Clockwork Orange"
<code>mdb:film/424</code>	<code>movie:director</code>	<code>mdb:director/8476</code>
<code>mdb:film/424</code>	<code>rdfs:label</code>	"Spartacus"
<code>mdb:actor/29704</code>	<code>movie:actor_name</code>	"Jack Nicholson"
<code>mdb:film/1267</code>	<code>movie:actor</code>	<code>mdb:actor/29704</code>
<code>mdb:film/1267</code>	<code>rdfs:label</code>	"The Last Tycoon"
<code>mdb:film/3418</code>	<code>movie:actor</code>	<code>mdb:actor/29704</code>
<code>mdb:film/3418</code>	<code>rdfs:label</code>	"The Passenger"
<code>geo:2635167</code>	<code>gn:name</code>	"United Kingdom"
<code>geo:2635167</code>	<code>gn:population</code>	62348447
<code>geo:2635167</code>	<code>gn:wikipediaArticle</code>	<code>wp:United_Kingdom</code>
<code>bm:books/0743424425</code>	<code>dc:creator</code>	<code>bm:persons/Stephen+King</code>
<code>bm:books/0743424425</code>	<code>rev:rating</code>	4.7
<code>bm:books/0743424425</code>	<code>scom:hasOffer</code>	<code>bm:offers/0743424425amazonOffer</code>
<code>lexvo:iso639-3/eng</code>	<code>rdfs:label</code>	"English"
<code>lexvo:iso639-3/eng</code>	<code>lvont:usedIn</code>	<code>lexvo:iso3166/CA</code>
<code>lexvo:iso639-3/eng</code>	<code>lvont:usesScript</code>	<code>lexvo:script/Latn</code>

Fig. 12.15 Example RDF dataset. Prefixes are used to identify the data sources

1. $V = V_c \cup V_e \cup V_l$ is a collection of vertices that correspond to all subjects and objects in RDF data, where V_c , V_e , and V_l are collections of class vertices, entity vertices, and literal vertices, respectively.
2. L_V is a collection of vertex labels.
3. A *vertex labeling function* $f_V : V \rightarrow L_V$ is a bijective function that assigns to each vertex a label. The label of a vertex $u \in V_l$ is its literal value, and the label of a vertex $u \in V_c \cup V_e$ is its corresponding URI.
4. $E = \{\overrightarrow{u_1, u_2}\}$ is a collection of directed edges that connect the corresponding subjects and objects.
5. L_E is a collection of edge labels.
6. An *edge labeling function* $f_E : E \rightarrow L_E$ is a bijective function that assigns to each edge a label. The label of an edge $e \in E$ is its corresponding property.

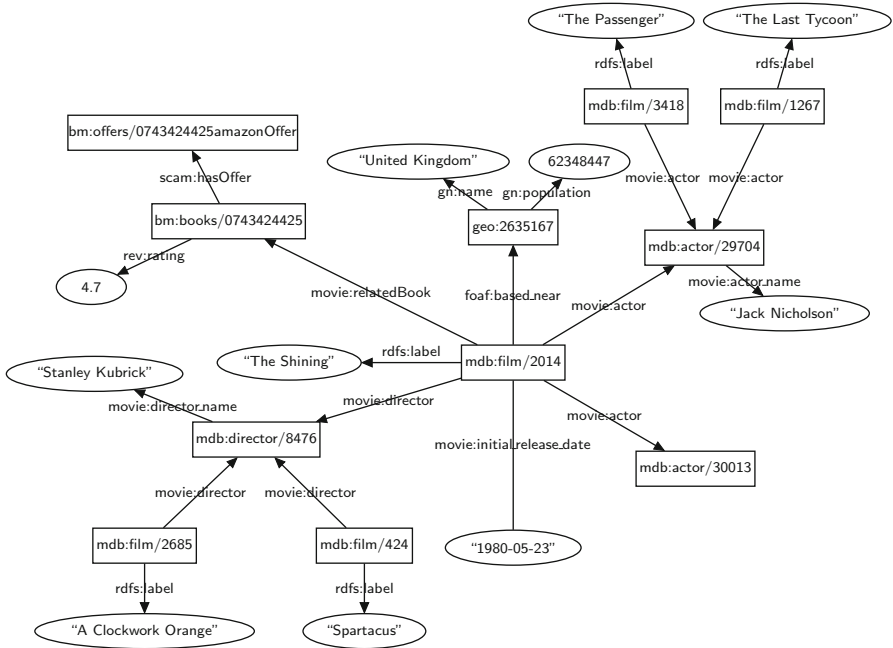


Fig. 12.16 RDF graph corresponding to the dataset in Fig. 12.15

An edge $\overrightarrow{u_1, u_2}$ is an *attribute property edge* if $u_2 \in V_l$; otherwise, it is a *link edge*.

Note that RDF graph structure is different than the property graphs we discussed in Chap. 10. As you will recall, property graphs have attributes attached to vertices and edges allowing sophisticated value-based predicates to be specified in queries. In RDF graphs, the only attribute of a vertex or an edge is the vertex/edge label. What would be vertex attributes in a property graph become edges whose labels are the attribute names. Therefore, RDF graphs are simpler and more regular, but generally larger in terms of the number of vertices and edges.

Figure 12.16 shows an example of an RDF graph. The vertices that are denoted by boxes are entity or class vertices, and the others are literal vertices.

The W3C standard language for RDF is SPARQL, which can be defined as follows [Hartig 2012]. Let \mathcal{U} , \mathcal{B} , \mathcal{L} , and \mathcal{V} denote the sets of all URIs, blank nodes, literals, and variables, respectively. A SPARQL expression is expressed recursively

1. A triple pattern $(\mathcal{U} \cup \mathcal{B} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L} \cup \mathcal{V})$ is a SPARQL expression,
2. (optionally) If P is a SPARQL expression, then $P \text{ FILTER } R$ is also a SPARQL expression where R is a built-in SPARQL filter condition,
3. (optionally) If P_1 and P_2 are SPARQL expressions, then $P_1 \text{ AND|OPT|OR } P_2$ are also SPARQL expressions.

A set of triple patterns is called *basic graph pattern* (BGP) and SPARQL expressions that only contain these are called *BGP queries*. These are the subject of most of the research in SPARQL query evaluation.

Example 12.15 An example SPARQL query that finds the names of the movies directed by “Stanley Kubrick” and have a related book that has a rating greater than 4.0 is specified as follows:

```

SELECT ?name
WHERE {
  ?m rdfs:label ?name. ?m movie:director ?d.
  ?d movie:director_name "Stanley Kubrick".
  ?m movie:relatedBook ?b. ?b rev:rating ?r.
  FILTER(?r > 4.0)
}

```

In this query, the first three lines in the **WHERE** clause form a BGP consisting of five triple patterns. All triple patterns in this example have *variables*, such as “?m”, “?name” and “?r”, and “?r” has a filter: **FILTER**(?r > 4.0). ◆

A SPARQL query can also be represented as a *query graph*. A *query graph* is a seven-tuple $Q = \langle V^Q, L_V^Q, E^Q, L_E^Q, f_V^Q, f_E^Q, FL \rangle$, where

1. $V^Q = V_c^Q \cup V_e^Q \cup V_l^Q \cup V_p^Q$ is a collection of vertices that correspond to all subjects and objects in a SPARQL query, where V_p^Q is a collection of variable vertices (corresponding to variables in the query expression), and V_c^Q and V_e^Q and V_l^Q are collections of class vertices, entity vertices, and literal vertices in the query graph Q , respectively.
2. E^Q is a collection of edges that correspond to properties in a SPARQL query.
3. L_V^Q is a collection of vertex labels in Q and L_E^Q is the edge labels in E^Q .
4. $f_V^Q : V^Q \rightarrow L_V^Q$ is a bijective vertex labeling function that assigns to each vertex in Q a label from L_V^Q . The label of a vertex $v \in V_p^Q$ is the variable; that of a vertex $v \in V_l^Q$ is its literal value; and that of a vertex $v \in V_c^Q \cup V_e^Q$ is its corresponding URI.
5. $f_E^Q : V^Q \rightarrow L_E^Q$ is a bijective vertex labeling function that assigns to each edge in Q a label from L_E^Q . An edge label can be a property or an edge variable.
6. FL are constraint filters.

The query graph for Q_1 is given in Fig. 12.17.

The semantics of SPARQL query evaluation can, therefore, be defined as subgraph matching using graph homomorphism whereby all subgraphs of an RDF graph G are found that are homomorphic to the SPARQL query graph Q .

It is usual to talk about SPARQL query types based on the shape of the query graph (we will refer to these types in the following discussion). Typically, three

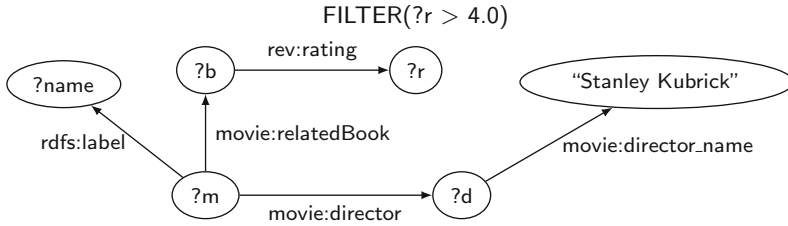


Fig. 12.17 SPARQL query graph corresponding to query Q_1

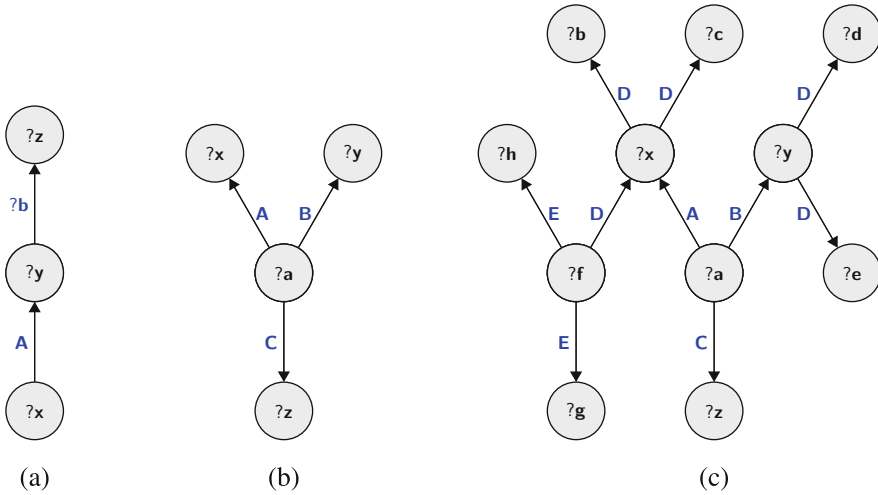


Fig. 12.18 Sample SPARQL query shapes. (a) Q_L . (b) Q_S . (c) Q_K

query types are observed: (i) linear (Fig. 12.18a), where the variable in the object field of one triple pattern appears in the subject of another triple pattern (e.g., $?y$ in Q_L) (ii) star-shaped (Fig. 12.18b), where the variable in the object field of one triple pattern appears in the subject of multiple other triple patterns (e.g., $?a$ in Q_S), and (iii) snowflake-shaped (Fig. 12.18c), which is a combination of multiple star queries.

A number of RDF data management systems have been developed. These can be broadly classified into five groups: those that map the RDF data directly into a relational system, those that use a relational schema with extensive indexing (and a native storage system), those that denormalize the triples table into clustered properties, those that use column-store organization, and those that exploit the native graph pattern matching semantics of SPARQL.

Direct Relational Mapping

Direct relational mapping systems take advantage of the fact that RDF triples have a natural tabular structure. Therefore, they create a single table with three columns (Subject, Property, Object) that holds the triples (there usually are additional auxiliary tables, but we ignore them here). The SPARQL query can then be translated into SQL and executed on this table. It has been shown that SPARQL 1.0 can be full translated to SQL; whether the same is true for SPARQL 1.1 with its added features is still open. This approach aims to exploit the well-developed relational storage, query processing and optimization techniques in executing SPARQL queries. Systems such as Sesame SQL92SAIL¹⁰ and Oracle follow this approach.

Example 12.16 Assuming that the table given in Fig. 12.15 is a relational table, the example SPARQL query in Example 12.15 can be translated to the following SQL query (where s,p,o correspond to column names: Subject, Property, Object):

```
SELECT T1.object
FROM T AS T1, T AS T2, T AS T3,
T AS T4, T AS T5
WHERE T1.p="rdfs:label"
AND T2.p="movie:relatedBook"
AND T3.p="movie:director"
AND T4.p="rev:rating"
AND T5.p="movie:director_name"
AND T1.s=T2.s
AND T1.s=T3.s
AND T2.o=T4.s
AND T3.o=T5.s
AND T4.o > 4.0
AND T5.o="Stanley Kubrick"
```



As can be seen from this example, this approach results in a high number of self-joins that are not easy to optimize. Furthermore, in large datasets, this single triples table becomes very large, further complicating query processing.

¹⁰Sesame is built to interact with any storage system since it implements a Storage and Inference Layer (SAIL) to interface with the particular storage system on which it sits. SQL92SAIL is the specific instantiation to work on relational systems.

Single Table Extensive Indexing

One alternative to the problems created by direct relational mapping is to develop native storage systems that allow extensive indexing of the triple table. Hexastore and RDF-3X are examples of this approach. The single table is maintained, but extensively indexed. For example, RDF-3X creates indexes for all six possible permutations of the subject, property, and object: (spo, sop, ops, ops, sop, pos). Each of these indexes is sorted lexicographically by the first column, followed by the second column, followed by the third column. These are then stored in the leaf pages of a clustered B⁺-tree.

The advantage of this type of organization is that SPARQL queries can be efficiently processed regardless of where the variables occur (subject, property, object) since one of the indexes will be applicable. Furthermore, it allows for index-based query processing that eliminates some of the self-joins—they are turned into range queries over the particular index. Even when joins are required, fast merge-join can be used since each index is sorted on the first column. The obvious disadvantages are, of course, the space usage, and the overhead of updating the multiple indexes if data is dynamic.

Property Tables

Property tables approach exploits the regularity exhibited in RDF datasets where there are repeated occurrence of patterns of statements. Consequently, it stores “related” properties in the same table. The first system that proposed this approach is Jena; IBM’s DB2RDF also follows the same strategy. In both of these cases, the resulting tables are mapped to a relational system and the queries are converted to SQL for execution.

Jena defines two types of property tables. The first type, which can be called *clustered property table*, group together the properties that tend to occur in the same (or similar) subjects. It defines different table structures for single-valued properties versus multivalued properties. For single-valued properties, the table contains the subject column and a number of property columns (Fig. 12.19a). The value for a given property may be null if there is no RDF triple that uses the subject and that property. Each row of the table represents a number of RDF triples—the same number as the nonnull property values. For these tables, the subject is the primary key. For multivalued properties, the table structure includes the subject and the multivalued property (Fig. 12.19b). Each row of this table represents a single RDF triple; the key of the table is the compound key (subject, property). The mapping of the single triple table to property tables is a database design problem that is done by a database administrator.

Jena also defines a *property class table* that cluster the subjects with the same *type* of property into one property table (Fig. 12.19c). In this case, all members of a class (recall our discussion of class structure within the context of RDFS) together

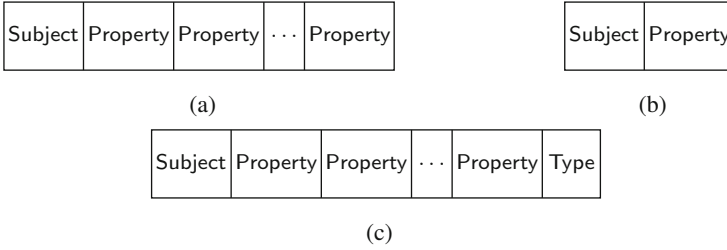


Fig. 12.19 Clustered property table design

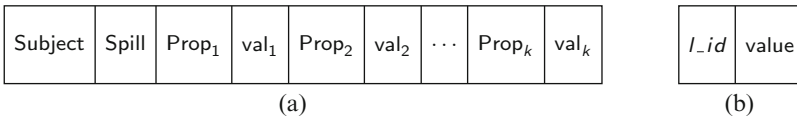


Fig. 12.20 DB2RDF table design. (a) DPH. (b) DS

in one table. The “Type” column is the value of `rdf:type` for each property in that row.

Example 12.17 The example dataset in Example 12.14 may be organized to create one table that includes the properties of subjects that are films, one table for properties of directors, one table for properties of actors, one table for properties of books and so on. ♦

IBM DB2RDF also follows the same strategy, but with a more dynamic table organization (Fig. 12.20). The table, called *direct primary hash* (DPH) is organized by each subject, but instead of manually identifying “similar” properties, the table accommodates k property columns, each of which can be assigned a different property in different rows. Each property column is, in fact, two columns: one that holds the property label, and the other that holds the value. If the number of properties for a given subject is greater than k , then the extra properties are spilled onto a second row and this is marked on the “spill” column. For multivalued properties, a *direct secondary hash* (DSH) table is maintained—the original property value stores a unique identifier *l_id*, which appears in the DS table along with the values.

The advantage of property table approach is that joins in star queries (i.e., subject-subject joins) become single table scans. Therefore, the translated query has fewer joins. The disadvantages are that in either of the two forms discussed above, there could be a significant number of null values in the tables, and dealing with multivalued properties requires special care. Furthermore, although star queries can be handled efficiently, this approach may not help much with other query types. Finally, when manual assignment is used, clustering “similar” properties is nontrivial and bad design decisions exacerbate the null value problem.

Binary Tables

Binary tables approach follows column-oriented database schema organization and defines a two-column table for each property containing the subject and object. This results in a set of tables each of which is ordered by the subject. This is a typical column-oriented database organization and benefits from the usual advantages of such systems such as reduced I/O due to reading only the needed properties and reduced tuple length, compression due to redundancy in the column values, etc. In addition, it avoids the null values that is experienced in property tables as well as the need for manual or automatic clustering algorithms for “similar” properties, and naturally supports multivalued properties—each becomes a separate row as in the case of Jena’s DS table. Furthermore, since tables are ordered on subjects, subject-subject joins can be implemented using efficient merge-joins. The shortcomings are that the queries require more join operations some of which may be subject-object joins that are not helped by the merge-join operation. Furthermore, insertions into the tables have higher overhead since multiple tables need to be updated. It has been argued that the insertion problem can be mitigated by batch insertions, but in dynamic RDF repositories the difficulty of insertions is likely to remain a significant problem. The proliferation of the number of tables may have a negative impact on the scalability (with respect to the number of properties) of binary tables approach.

Example 12.18 For example, the binary table representation of the dataset given in Example 12.14 would create one table for each unique property—there are 18 of them. Two of these tables are shown in Fig. 12.21. ♦

Graph-Based Processing

Graph-based RDF processing approaches fundamentally implement the semantics of RDF queries as defined at the beginning of this section. In other words, they maintain the graph structure of the RDF data (using some representation such as adjacency lists), convert the SPARQL query to a query graph, and do subgraph

Subject	Object
film/2014	“The Shining”
film/2685	“A Clockwork Orange”
film/424	“Spartacus”
film/1267	“The Last Tycoon”
film/3418	“The Passenger”
iso639-3/eng	“English”

(a)

Subject	Object
film/2014	actor/29704
film/2014	actor/30013
film/1267	actor/29704
film/3418	actor/29704

(b)

Fig. 12.21 Binary table organization of properties (a) “rdfs:label” and (b) “movie:actor” from the example dataset (prefixes are removed)

matching using homomorphism to evaluate the query against the RDF graph. Systems such as gStore, and chameleon-db follow this approach.

The advantage of this approach is that it maintains the original representation of the RDF data and enforces the intended semantics of SPARQL. The disadvantage is the cost of subgraph matching—graph homomorphism is NP-complete. This raises issues with respect to the scalability of this approach to large RDF graphs; typical database techniques including indexing can be used to address this issue. In the remainder, we present the approach within the context of the gStore system to highlight the issues.

gStore uses adjacency list representation of graphs. It encodes each entity and class vertex into a fixed length bit string that captures the “neighborhood” information for each vertex and exploits this during graph matching. This results in the generation of a *data signature graph* G^* , in which each vertex corresponds to a class or an entity vertex in the RDF graph G . Specifically, G^* is induced by all entity and class vertices in the original RDF graph G together with the edges whose endpoints are either entity or class vertices. Figure 12.22a shows the data signature graph G^* that corresponds to RDF graph G in Fig. 12.16. An incoming SPARQL query is also represented as a *query graph* Q that is similarly encoded into a *query*

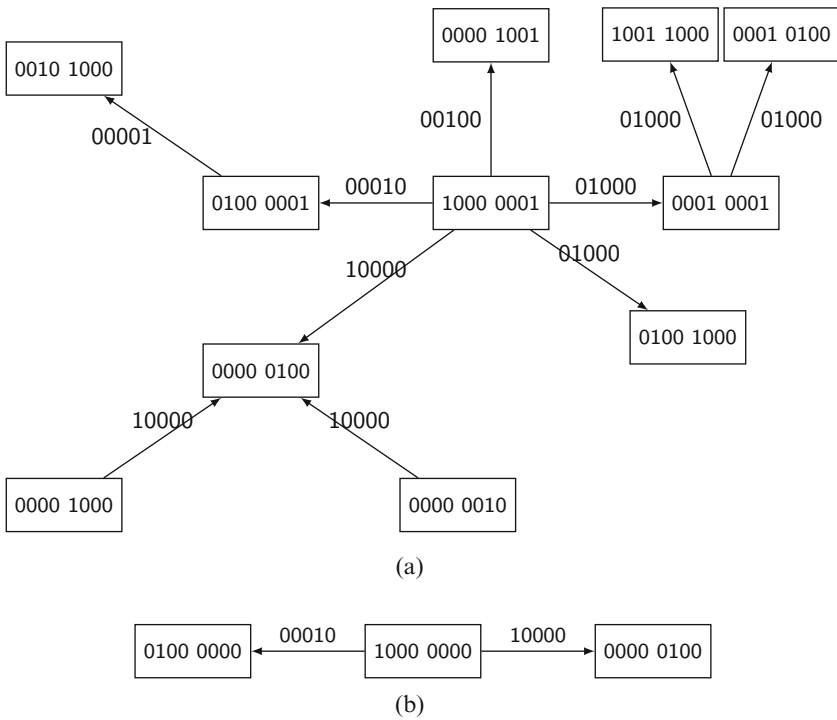


Fig. 12.22 Signature graphs. (a) Data signature graph G^* . (b) Query signature graph Q^*

signature graph Q^* . The encoding of query graph depicted in Fig. 12.17 into a query signature graph Q_2^* is shown in Fig. 12.22b.

The problem now turns into finding matches of Q^* over G^* . Although both the RDF graph and the query graph are smaller as a result of encoding, the NP-completeness of the problem remains. Therefore, gStore uses a filter-and-evaluate strategy to reduce the search space over which matching is applied. The objective is to first use a false-positive pruning strategy to find a set of candidate subgraphs (denoted as CL), and then validate these using the adjacency list to find answers (denoted as RS). Accordingly, two issues need to be addressed. First, the encoding technique should guarantee that $RS \subseteq CL$ —the encoding described above provably achieves this. Second, an efficient subgraph matching algorithm is required to find matches of Q^* over G^* . For this, gStore uses an index structure called VS^* -tree that is a summary graph of G^* . VS^* -tree is used to efficiently process queries using a pruning strategy to reduce the search space for finding matches of Q^* over G^* .

Distributed and Federated SPARQL Execution

As RDF collections grow, scale-out solutions to scaling have been developed involving parallel and distributed processing. Many of these solutions divide an RDF graph G into several fragments and place each at a different site in a parallel/distributed system. Each site hosts a centralized RDF store of some kind. At runtime, a SPARQL query Q is decomposed into several subqueries such that each subquery can be answered locally at one site, and the results are then aggregated. Each of these papers proposes its own data partitioning strategy, and different partitioning strategies result in different query processing methods. Some of the approaches use MapReduce-based solutions where RDF triples are stored in HDFS and each triple pattern is evaluated by scanning the HDFS files followed by a MapReduce join implementation. Other approaches follow more or less the distributed/parallel query processing methodologies described in detail in various chapters of this book whereby the query is partitioned into subqueries and evaluated across the sites.

An alternative that has been proposed is to use partial query evaluation for executing distributed SPARQL queries. Partial function evaluation is a well-known programming language strategy whose basic idea is the following: given a function $f(s, d)$, where s is the known input and d is the yet unavailable input, the part of f 's computation that depends only on s generates a partial answer. In this approach, data is partitioned, but queries are not—each site receives the full SPARQL query Q and executes it on the local RDF graph fragment providing data parallel computation. In this particular setting, the partial evaluation strategy is applied as follows: each site S_i treats fragment F_i as the known input in the partial evaluation stage; the unavailable input is the rest of the graph ($\bar{G} = G \setminus F_i$). There are two important issues to be addressed in this framework. The first is to compute the partial evaluation results at each site S_i given a query graph Q —in other words, addressing the graph homomorphism of Q over F_i ; this is called the *local partial match* since

it finds the matches internal to fragment F_i . Since ensuring edge disjointness is not possible in vertex-disjoint partitioning, there will be *crossing edges* between graph fragments. The second task is the assembly of these local partial matches to compute crossing matches. This assembly task can be executed either on a control site or similar to distributed join.

The above approaches take a centralized RDF dataset and partition it for distributed/parallel execution. In many RDF settings, concerns arise similar to what we discussed in database integration requiring a federated solution. In the RDF world, some of the sites that host RDF data also have the capability to process SPARQL queries; these are called *SPARQL endpoints*. A typical example is LOD, where different RDF repositories are interconnected, providing a *virtually integrated distributed database*. A common technique in federated RDF environments is to precompute metadata for each individual SPARQL endpoint. The metadata can specify the capabilities of the end point or a description of the triple patterns (i.e., property) that can be answered at that endpoint, or other information that the particular algorithm uses. Based on the metadata, the original SPARQL query is decomposed into several subqueries, where each subquery is sent to its relevant SPARQL endpoints. The results of subqueries are then joined together to answer the original SPARQL query.

An alternative to precomputing metadata is to make use of SPARQL ASK queries to gather information about each endpoint and to construct the metadata on the fly. Based on the results of these queries, a SPARQL query is decomposed into subqueries and assigned to endpoints.

12.6.2.3 Navigating and Querying the LOD

LOD consists of a set of *web documents*. The starting point, therefore, is a web document with embedded RDF triples that encode web resources. The RDF triples contain *data links* to other documents that allow web documents to be interconnected to get the graph structure.

The semantics of SPARQL queries over the LOD becomes tricky. One possibility is to adopt *full web semantics* that specifies the scope of evaluating a SPARQL query expression to be all linked data. There is no known (terminating) query execution algorithm that can guarantee result completeness under this semantics. The alternative is a family of *reachability-based semantics* that define the scope of evaluating a SPARQL query in terms of the documents that can be reached: given a set of seed URIs and a reachability condition, the scope is all data along the paths of the data links from the seeds and that satisfy the reachability condition. The family is defined by different reachability conditions. In this case, there are computationally feasible algorithms.

There are three approaches to SPARQL query execution over LOD: traversal-based, index-based, and hybrid. *Traversal approaches* basically implement a reachability-based semantics: starting from seed URIs, they recursively discover relevant URIs by traversing specific data links at query execution runtime. For these

algorithms, the selection of the seed URIs is critical for performance. The advantage of traversal approaches is their simplicity (to implement) since they do not need to maintain any data structures (such as indexes). The disadvantages are the latency of query execution since these algorithms “browse” web documents, and repeated data retrieval from each document introduces significant latency. They also have limited possibility for parallelization—they can be parallelized to the same extent that crawling algorithms can.

The *index-based approaches* use an index to determine relevant URIs, thereby reducing the number of linked documents that need to be accessed. A reasonable index key is triple patterns in which case the “relevant” URIs for a given query are determined by accessing the index, and the query is evaluated over the data retrieved by accessing those URIs. In these approaches, data retrieval can be fully parallelized, which reduces the negative impact of data retrieval on query execution time. The disadvantages of the approach are the dependence on the index—both in terms of the latency that index construction introduces and in terms of the restriction the index imposes on what can be selected—and the freshness issues that result from the dynamicity of the web and the difficulty of keeping the index up-to-date.

Hybrid approaches perform a traversal-based execution using prioritized listing of URIs for look-up. The initial seeds come from a prepopulated index; new discovered URIs that are not in the index are ranked according to number of referring documents.

12.6.3 Data Quality Issues in Web Data Integration

In Chap. 7 (specifically in Sect. 7.1.5) we discussed data quality and data cleaning issues in the case of database integration (mainly data warehousing) systems. Data quality issues in web data are only more severe due to the sheer number of web data sources, the uncontrolled data entry process of web information sources, and the increased data diversity. Data quality encompasses both data consistency and veracity (authenticity and conformity of data with reality). In a data warehouse, data consistency is obtained through data cleaning, which deals with detecting and removing errors and inconsistencies from data. Data cleaning in the web context (and also in data lakes) is made difficult by the lack of schema information and the limited number of integrity constraints that can be defined without a schema.

Checking for data veracity remains a big challenge. However, if many different data sources overlap, as it is often the case with data coming from the web for instance, there will be a high-level of redundancy. It may be possible to use efficient data fusion techniques (to be discussed shortly) to detect the correct values for the same data items, and thus discover the truth.

In this section, we highlight some of the main data quality and data cleaning issues in web data and discuss current solutions for addressing them.

12.6.3.1 Cleaning Structured Web Data

Structured data represents an important category of data on the web, and they suffer from numerous data quality issues. In the following, we first summarize the techniques proposed in cleaning structured data in general. Then, we point out the unique challenges in cleaning structured data on the web.

Figure 12.23 shows a typical workflow for cleaning structured data, consisting of an optional discovery and profiling step, an error detection step, and an error repair step. To clean a dirty dataset, we often need to model various aspects of this data (metadata), e.g., schema, patterns, probability distributions, and other metadata. One way to obtain such metadata is by consulting domain experts, which is usually a costly and a time-consuming process, hence a discovery and profiling step is often used to discover these metadata automatically. Given a dirty dataset and the associated metadata, the error detection step finds part of the data that do not conform to the metadata, and declares this subset to contain errors. The errors surfaced by the error detection step can be in various forms, such as outliers, violations of integrity constraints, and duplicates. Finally, the error repair step produces data updates that are applied to the dirty dataset to remove detected errors. Since there are many uncertainties in the data cleaning process, external sources such as knowledge bases and human experts are consulted whenever possible to ensure the accuracy of the cleaning workflow.

The above process works well for structured tables that have a rich set of metadata, e.g., large schema with enough constraints to model columns and rows interactions. Also, the cleaning and error detection process works better when there are enough examples (tuples) for automatic algorithms to compare various instances to detect possible errors, and to leverage the redundancy in the data to correct these

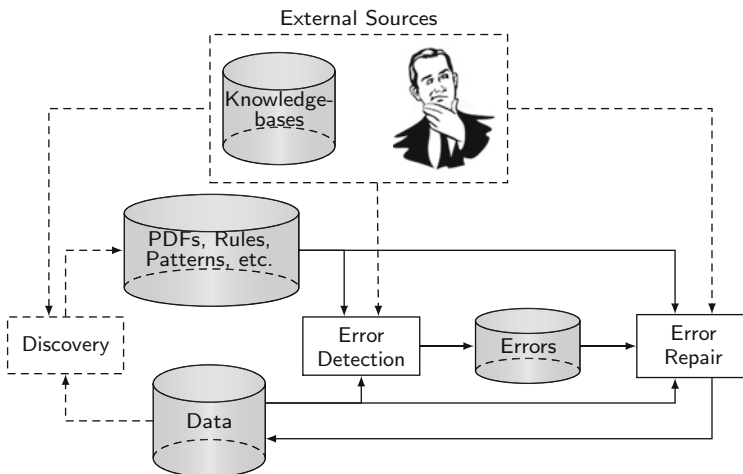


Fig. 12.23 A typical workflow for cleaning structured data

Sevilla - Jerez de la Frontera-Cádiz	1861
Córdoba - Málaga	1865.
Bobadilla - Granada	1874
Córdoba - Bélmez	1874
Osuna	La Roda

(a)

Polaco	15.04.1983	194	84
Vini	29.09.1982	N/A	N/A
Caiao	30/11/1982	N/A	N/A
Jairo	17.02.1990	N/A	N/A
Michael	20.04.1983	N/A	N/A
Ricardinho	19.11.1975	192	94

(b)

2002 ^[12]	10.300 oz	899,500 oz
2005 ^[13]	25.272	2.174.620 oz
2006 ^[13]	49.354 oz	3.005.611 oz
2007 ^[13]	48.807 oz	3.165408 oz
2008 ^[9]	47.755 oz	3.157.837 oz
2009 ²	0.9 million oz	818.050 oz

(c)

WARRIORS@Susses Thunder	13-28	—
WARRIORS@Hampshire Thrashers	42-13	—
Essex Spartans@WARRIORS	P-P	Postponed
WARRIORS@Cambridgeshire Cats	36-44	—
East Kent Mavericks@WARRIORS	12-18	—
WARRIORS@East Kent Mavericks	15-17	—

(d)

Fig. 12.24 Data quality issues on structured web data (erroneous data is marked in red cells). Adapted from [Huang and He 2018]. (a) Extra dot. (b) Mixed dates. (c) Inconsistent weights. (d) Score placeholder

errors. However, in web tables, both of these premises are not satisfied, as most of the tables are short (few tuples) and skinny (limited number of attributes). o make matters worse, the number of web tables is far greater than the number of tables in a data warehouse. This means that manual cleaning, though relatively for a single web table, is not feasible for all structured web tables. Figure 12.24 shows some sample errors found on Wikipedia tables, and there is an estimated 300K such errors.

12.6.3.2 Web Data Fusion

A common problem that arises often in web data integration is data fusion, namely deciding what is the correct value for an item that has different representations from multiple web sources. The problem is that different web sources can provide conflicting representations, and thus making data fusion hard. There are two types of data conflicts: *uncertainty* and *contradiction*. Uncertainty is a conflict between a nonnull value and one or more null values that are used to describe the same property of a real world entity. Uncertainty is caused by missing information, usually represented by null values in a source. Contradiction is a conflict between two or more different nonnull values that represent different values of the same property of a real world entity. Contradiction is caused by different sources providing different values for the same attribute.

Automatic cleaning of web tables is thus particularly challenging. While cleaning techniques developed in the context of data warehouse can be applied to clean some

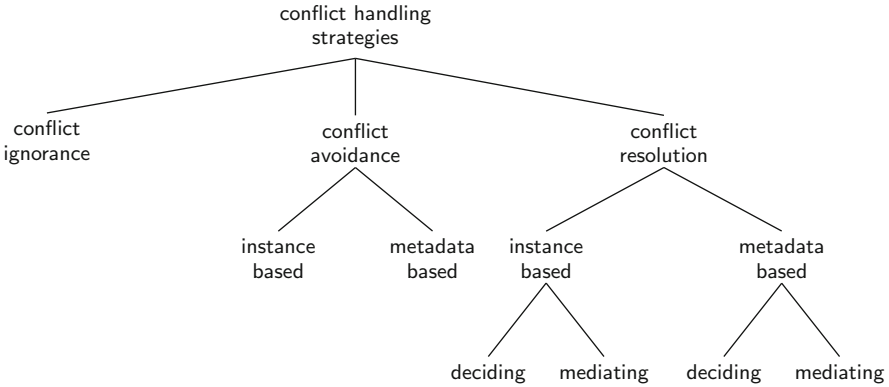


Fig. 12.25 Classification of strategies for data fusion. From [Bleiholder and Naumann 2009]

errors, cleaning web tables deserve more dedicated techniques. Auto-Detect is a recent proposal that aims at detecting such errors on web tables. Auto-Detect is a data-driven statistics-based techniques that leverage value co-occurrence statistics from large corpora for error detection. The main assumption is that if a certain value combination is extremely rare (quantified using point-wise mutual information), then it suggests a potential error. While Auto-Detect is able to detect many errors, it does not suggest data fixes. We have yet to see proposals that automatically repair errors (or even suggest fixes) in web table.

Figure 12.25 shows the classification of different data fusing strategies. *Conflict ignorance* strategies ignore the conflicts and simply pass the conflicts to the users or applications. *Conflict avoidance* strategies acknowledge the existence of conflicting representations, and apply a simple rule to take a unique decision based on either the data instance or the metadata. An example of instance based conflict avoidance strategy is to prefer nonnull values over null values. An example of metadata based conflict avoidance strategy is to prefer values from one source over values from another. *Conflict resolution* strategies resolve the conflicts, by picking a value from the already present values (deciding) or by choosing a value that does not necessarily exist among present values (mediating). An example of instance based, deciding conflict resolution strategy is to take the most frequent value. An example of instance based, mediating conflict resolution strategy is to take the average of all present values.

12.6.3.3 Web Source Quality

These basic conflict resolution strategies described above mostly rely on participating values to resolve conflicts, and they can fall short in the following three aspects. First, web sources have different qualities; data values provided by more accurate web sources are usually more accurate. However, more accurate web

sources can also provide incorrect values, therefore, an advanced resolution strategy is often needed to take source quality into consideration when predicting the correct value. Second, web sources can copy from each other, and ignoring these kinds of dependencies between web sources can cause wrong resolution decisions. For example, the majority vote strategy to resolve conflicts would be affected if some data items in a source are copied. Third, the correct value for a data item may evolve over time as well (e.g., a person's affiliation), hence, it is therefore crucial to distinguish between incorrect value and *outdated* value when evaluating source accuracies and making resolution decisions.

The building block of advanced data fusion strategies is to evaluate the trustworthiness or quality of a source. In this section, we discuss how the accuracy of a data source is modeled, and we mention how that model is extended to handle source dependencies and source freshness.

Source Accuracy

The accuracy of a source can be measured as the fraction of true values provided by a source. The accuracy of a source S is denoted by $A(S)$, which can be considered as the probability that a value provided by S is the true value. Let $V(S)$ denote the values provided by S . For each $v \in V(S)$, let $Pr(v)$ denote the probability that v is the true value. Then $A(S)$ is computed as follows:

$$A(S) = Avg_{v \in V(S)} Pr(v)$$

Consider a data item D . Let $Dom(D)$ be the domain of D , including one true value and n false values. Let S_D be the set of sources that provide a value for D , and let $S_D(v) \subseteq S_D$ be the set of sources that provide the value v for D . Let $\Phi(D)$ denote the observation of which value each $S \in S_D$ provides for D . The probability $Pr(v)$ can be computed as follows:

$$Pr(v) = Pr(v \text{ is true value} | \Phi(D)) \propto Pr(\Phi(D) | v \text{ is true value})$$

Assume that sources are independent and that the n false values are equally likely to happen, $Pr(\Phi(D) | v \text{ is true value})$ can be computed as follows:

$$Pr(\Phi(D) | v \text{ is true value}) = \prod_{S \in S_D(v)} A(S) \prod_{S \in S_D \setminus S_D(v)} \frac{1 - A(S)}{n}$$

which can be rewritten as

$$Pr(\Phi(D) | v \text{ is true value}) = \prod_{S \in S_D(v)} \frac{nA(S)}{1 - A(S)} \prod_{S \in S_D} \frac{1 - A(S)}{n}$$

Since $\prod_{S \in S_D} \frac{1-A(S)}{n}$ is the same for all values, we have

$$Pr(\Phi(D)|v \text{ is true value}) \propto \prod_{S \in S_D(v)} \frac{nA(S)}{1-A(S)}$$

Accordingly, the *vote count* of a data source S is defined as:

$$C(S) = \ln \frac{nA(S)}{1-A(S)}$$

The *vote count* of a value v is defined as:

$$C(v) = \sum_{S \text{ in } S_D(v)} C(S)$$

Intuitively, a source with a higher vote count is more accurate and a value with a higher vote count is more likely to be true. Combining the above analysis, the probability of each value v can be computed as follows:

$$Pr(v) = \frac{\exp(C(v))}{\sum_{v_0 \text{ in } Dom(v)} \exp(C(v_0))}$$

Obviously, for a data item D , the value $v \in Dom(D)$ with the highest probability $Pr(v)$ would be selected as the true value. As we can see, the computation of the source accuracy $A(S)$ depends on the probability $Pr(v)$, and the computation of the probability $Pr(v)$ depends on the source accuracy $A(S)$. An algorithm is possible that starts with the same accuracy for every source and the same probability for every value, and iteratively computes probabilities for all sources and probabilities for all values until convergence. The convergence criteria is set to be when there is no change in source accuracies and no oscillation in decided true values.

Source Dependency

The above computation for source accuracy assumes that sources are independent. In reality, sources copy from each other, which creates dependencies. There are two intuitions for copy detection between sources. First, for a particular data item, there is only one true value, but there are usually multiple false values. Two sources sharing the same true value does not necessarily imply dependency; however, two sources sharing the same false value is typically a rare event, and thus would more likely imply source dependency. Second, a random subset of values provided by a data source would typically have similar accuracies as the full set of values provided by the data source. However, for a copier data source, the subset of values it copies may have different accuracies than the rest of the values it provides independently.

Thus, between two dependent sources where one copies another, the source whose own data values' accuracies differ significantly from the values shared with the other source is more likely to be the copier. Based on these intuitions, a Bayesian model can be developed to compute the probability of copying between two sources S_1 and S_2 given the observations Φ on all data items; this probability is then used to adjust the computation of the vote count for a value $C(v)$ to account for source dependencies.

Source Freshness

We have so far assumed that data fusion is done on a static snapshot of the data. However, in reality, data evolves over time and the true value for an item might change as well. For example, the scheduled departure time for a flight might change in different months; a person's affiliation might change over time; and the CEO of a company could also change. To capture such changes, data sources will need to update their data. In this dynamic setting, data errors occur for these several reasons: (1) the sources may provide wrong values, similar to the static setting; (2) the sources may fail to update their data at all; and (3) some sources may not update their data in time. Data fusion, in this context, aims at finding all correct values and their valid periods in the history, when the true values evolve over time. While the source quality can be captured by accuracy in the static case, the metrics for evaluating source quality are more complicated in the dynamic setting—a high-quality source should provide a new value for a data item *if and only if, and right after* the value becomes the true value. Three metrics can be used to capture this intuition: the *coverage* of a source measures the transitions of different data items that it captures; the *exactness* measures the percentage of transitions a source mis-captures (by providing a wrong value); and the *freshness* measures how quickly a value change is captured by a source. Again, it is possible to rely on Bayesian analysis to decide both the time and the value of each transition for a data item.

Machine learning and probabilistic models have also been used in data fusion and modeling data source quality. In particular, SLiMFAST is a framework that expresses data fusion as a statistical learning problem over discriminative probabilistic models. In contrast to previous learning-based fusion approaches, SLiMFAST provides quality guarantees for the fused results, and it can also incorporate available domain knowledge in the fusion process. Figure 12.26 provides the system overview of SLiMFAST. The input to SLiMFAST includes (1) a collection of source observations, namely the possibly conflicting values provided for different objects by different sources; (2) an optional set of labeled ground truth, namely the true values for a subset of objects; and (3) some domain knowledge about sources that users deem to be informative of the accuracies of data sources. SLiMFAST takes all of these information, and compiles them into a probabilistic graphical model for holistic learning and inference. Depending on the how much ground truth data is available, SLiMFAST will decide which algorithm (expectation-maximization or empirical loss minimization) to use for learning the parameters of the graphical models. The

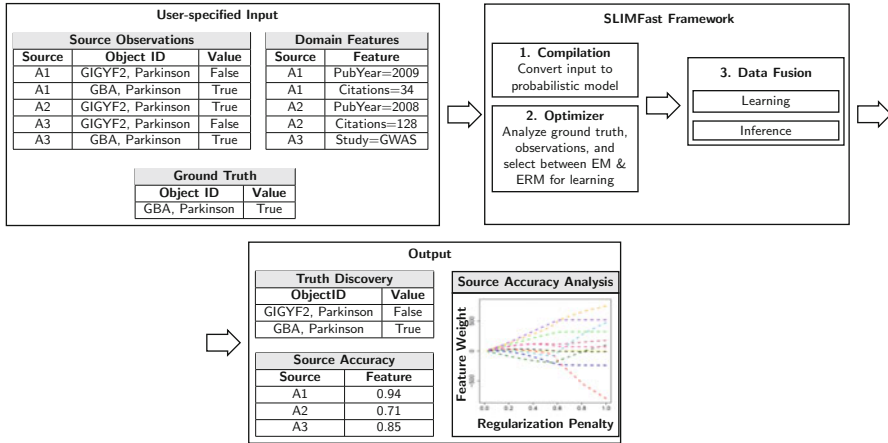


Fig. 12.26 Overview of SLiMFast. From [Rekatsinas et al. 2017]

learned model is then used for inferring both the value of objects and the source accuracies, as shown in the output.

12.7 Bibliographic Notes

There are a number of good sources on web topics, each with a slightly different focus. Abiteboul et al. [2011] focus on the use of XML and RDF for web data modeling and also contain discussions of search, and big data technologies such as MapReduce. A web data warehousing perspective is given in [Bhowmick et al. 2004]. Bonato [2008] primarily focuses on the modeling of the web as a graph and how this graph can be exploited. Early work on the web query languages and approaches are discussed in [Abiteboul et al. 1999].

A very good overview of web search issues is [Arasu et al. 2001], which we also follow in Sect. 12.2. Additionally, Lawrence and Giles [1998] provides an earlier discussion on the same topic focusing on the open web. Florescu et al. [1998] survey web search issues from a database perspective. Deep (hidden) web is the topic of [Raghavan and Garcia-Molina 2001]. Lage et al. [2002] and Hedley et al. [2004b] also discuss search over the deep web and the analysis of the results. Metasearch for accessing the deep web is discussed in [Ipeirotis and Gravano 2002, Callan and Connell 2001, Callan et al. 1999, Hedley et al. 2004a]. The metasearch-related problem of database selection is discussed by Ipeirotis and Gravano [2002] and Gravano et al. [1999] (GLOSS algorithm).

Statistics about the open web are taken from [Bharat and Broder 1998, Lawrence and Giles 1998, 1999, Gulli and Signorini 2005] and those related to the deep web are due to [Hirate et al. 2006] and [Bergman 2001].

The graph structure of the web and using graphs to model and query the web is the topic of many publications: [Kumar et al. 2000, Raghavan and Garcia-Molina 2003, Kleinberg et al. 1999] discuss web graph modeling, [Kleinberg et al. 1999, Brin and Page 1998, Kleinberg 1999] focus on graphs for search, and [Chakrabarti et al. 1998] for categorization and classification of web content. The discussion on the characteristics of the web graph and its bow-tie structure are due to Bonato [2008], Broder et al. [2000] and Kumar et al. [2000]. We did not discuss the important issues related to the management of the very large, dynamic, and volatile web graph. These are beyond the scope of this chapter, but two lines of research can be identified. The first one compresses the web graph for more efficient storage and manipulation [Adler and Mitzenmacher 2001], while the second one suggests a special representation for the web graph called S-nodes [Raghavan and Garcia-Molina 2003].

Issues on web crawling are the subject of [Cho et al. 1998, Najork and Wiener 2001] and [Page et al. 1998], the latter being the classical paper on PageRank whose revised form as discussed in this chapter is due to Langville and Meyer [2006]. Alternative crawling approaches are the subjects of [Cho and Garcia-Molina 2000] (change frequency-based), [Cho and Ntoulas 2002] (sampling-based) and [Edwards et al. 2001] (incremental). Classification techniques for evaluating relevance are discussed by [Mitchell 1997, Chakrabarti et al. 2002] (naïve Bayes), Passerini et al. [2001], Altingövdé and Ulusoy [2004] (extensions of Bayesian), and by McCallum et al. [1999], Kaelbling et al. [1996] (reinforcement learning).

Web indexing is an important issue that we discussed in Sect. 12.2.2. Various text indexing methods are discussed in [Manber and Myers 1990] (suffix arrays), [Hersh 2001] [Lim et al. 2003] (inverted indexes), and [Faloutsos and Christodoulakis 1984] (signature files). Salton [1989] is probably the classical source for text processing and analysis. The challenges of building inverted indexes for the web and solutions are discussed by Arasu et al. [2001], Melnik et al. [2001], and Ribeiro-Neto and Barbosa [1998]. Related to this, ranking has been the topic of extensive research. In addition to the well-known PageRank, the HITS algorithm is due to Kleinberg [1999].

In our discussion of semistructured data approach to web querying we highlighted OEM data model and the Lorel language to expose the concepts. These are discussed in [Papakonstantinou et al. 1995] and [Abiteboul et al. 1997]. The data guides to simplify OEM are discussed in [Goldman and Widom 1997]. UnQL [Buneman et al. 1996] has similar concepts to Lorel. Our discussion of web query languages in Sect. 12.3.2 separated the languages into first and second generation; this is due to Florescu et al. [1998]. First generation languages include WebSQL [Mendelzon et al. 1997], W3QL [Konopnicki and Shmueli 1995], and WebLog [Lakshmanan et al. 1996]. The second generation languages include WebOQL [Arocena and Mendelzon 1998], and StruQL [Fernandez et al. 1997]. In the Query-Answering approach we referred to a number of systems: Mulder [Kwok et al. 2001], WebQA [Lam and Özsu 2002], Start [Katz and Lin 2002], and Tritus [Agichtein et al. 2004].

The components of the semantic web is presented by Antoniou and Plexousakis [2018]. The Linked Open Data (LOD) vision and its requirements are discussed by Bizer et al. [2018] and Berners-Lee [2006]. The topical domain separation of LOD is outlined in [Schmachtenberg et al. 2014].

Our discussion of RDF is primarily based on [Özsu 2016]. Five main approaches are described to managing RDF data: (1) direct relational mapping—Angles and Gutierrez [2008], Sequeda et al. [2014] discuss mapping SPARQL to SQL, Broekstra et al. [2002], and Chong et al. [2005] discuss Sesame SQL92SAIL and Oracle, respectively; (2) using a single table with extensive indexing (Hexastore [Weiss et al. 2008] and RDF-3X [Neumann and Weikum 2008, 2009]); (3) property tables (Jena [Wilkinson 2006]; IBM’s DB2RDF [Bornea et al. 2013]); (4) binary tables (SW-Store [Abadi et al. 2009] based on the proposal by Abadi et al. [2007]) whose problems in terms of table proliferation is discussed in [Sidirourgos et al. 2008]; (5) graph-based ([Bönström et al. 2003], gStore [Zou et al. 2011, 2014], and chameleon-db [Aluç 2015]). Graph-based techniques are discussed in detail by Zou and Özsu [2017]. The distributed and cloud-based SPARQL execution is addressed in [Kaoudi and Manolescu 2015]. Three approaches are identified for SPARQL query execution over LOD [Hartig 2013a]: traversal-based [Hartig 2013b, Ladwig and Tran 2011], index-based [Umbrich et al. 2011], and hybrid [Ladwig and Tran 2010].

Cleaning structured data has been extensively studied in warehouse integration settings [Rahm and Do 2000] and [Ilyas and Chu 2015]. Expansion to a broader context, including the web, is covered in Ilyas and Chu [2019]. In our discussion of data fusion (Sect. 12.6.3.2), the separation of data conflicts into *uncertainty* and *contradiction* is due to Dong and Naumann [2009]. In the same section, the discussion of Auto-Detect is due to [Huang and He 2018] and the classification discussion (as well as Fig. 12.25) is from [Bleiholder and Naumann 2009]. The discussion of data source modeling accuracy in Sect. 12.6.3.3, its extension to handle source dependencies and source freshness are due to Dong et al. [2009b,a]. For a more comprehensive treatment on the subject of advanced data fusion, we refer readers to the tutorial [Dong and Naumann 2009] and the book [Dong and Srivastava 2015]. The SlimFAST system (see Fig. 12.26) is presented in [Rekatsinas et al. 2017] and [Koller and Friedman 2009].

One of the first systems to deal with data cleaning issues in data lakes is CLAMS [Farid et al. 2016], which allows discovering and enforcing integrity constraints over a data lake’s data. CLAMS uses a graph data model, based on RDF, and a new integrity constraint formalism to capture both relational constraints and more expressive quality rules based on graph patterns as *denial constraints* [Chu et al. 2013]. CLAMS also uses Spark and parallel algorithms to enforce the constraints and detect data inconsistencies.

Exercises

Problem 12.1 How does web search differ from web querying?

Problem 12.2 ()** Consider the generic search engine architecture in Fig. 12.2. Propose an architecture for a web site with a shared-nothing cluster that implements all the components in this figure as well as web servers in an environment that will support very large sets of web documents and very large indexes, and very high numbers of web users. Define how web pages in the page directory and indexes should be partitioned and replicated. Discuss the main advantages of your architecture with respect to scalability, fault-tolerance, and performance.

Problem 12.3 ()** Consider your solution in Problem 12.2. Now consider a keyword search query from a web client to the web search engine. Propose a parallel execution strategy for the query that ranks the result web pages, with a summary of each web page.

Problem 12.4 (*) To increase locality of access and performance in different geographical regions, propose an extension of the web site architecture in Problem 12.3 with multiple sites, with web pages being replicated at all sites. Define how web pages are replicated. Define also how a user query is routed to a web site. Discuss the advantages of your architecture with respect to scalability, availability and performance.

Problem 12.5 (*) Consider your solution in Problem 12.4. Now consider a keyword search query from a web client to the web search engine. Propose a parallel execution strategy for the query that ranks the result web pages, with a summary of each web page.

Problem 12.6 ()** Consider two web data sources that we model as relations EMP1(Name, City, Phone) and EMP2(Firstname, Lastname, City). After schema integration, assume the view EMP(Firstname, Name, City, Phone) defined over EMP1 and EMP2, where each attribute in EMP comes from an attribute of EMP1 or EMP2, with EMP2. Lastname being renamed as Name. Discuss the limitations of such integration. Now consider that the two web data sources are XML. Give a corresponding definition of the XML schemas of EMP1 and EMP2. Propose an XML schema that integrates EMP1 and EMP2, and avoids the problems identified with EMP.