Chapter 17
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. According to two studies in 1998, there were 200 million [Bharat and Broder, 1998] to upwards of 320 million [Lawrence and Giles, 1998] static web pages. A 1999 study reported the size of the web as 800 million pages [Lawrence and Giles, 1999]. By 2005, the number of pages were reported to be 11.5 billion [Gulli and Signorini, 2005]. Today it is estimated that the web contains over 25 billion pages[1] and growing. These are numbers for the "static" web pages, i.e., those whose content do not change unless the page owners make explicit changes. The size of the web is much larger when "dynamic" web pages (i.e., pages whose content changes based on the context of user requests) are considered. A 2005 study reported the size to be over 53 billion pages [Hirate et al., 2006]. Additionally, it was estimated that, as of 2001, over 500 billion documents existed in the *deep web* (which we define below) [Bergman, 2001]. Besides its size, the web is very dynamic and changes rapidly. Thus, 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) [Lawrence and Giles, 1998]. This is composed of all static (and cross-linked) web pages that exist on web servers. The other component, which is known as the *hidden web* [Florescu et al., 1998] (or the *deep web* [Raghavan and Garcia-Molina, 2001]), 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.

The difference between the two 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

[1] See http://www.worldwidewebsize.com/

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_17, © Springer Science+Business Media, LLC 2011

searching the indexed data (as we discuss at length in Section 17.2). 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 Section 17.3.4).

Research on web data management has followed different threads. Most of the earlier work focused on keyword search and search engines. The subsequent 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. Along another front, XML has emerged as an important data format for representing data on the web. Thus, XML data management, and more recently *distributed* XML data management, have been topics of interest. 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 modelled as a graph. Both the structure of this graph and its management are important. This is discussed in Section 17.1. Web search is discussed in Section 17.2 and web querying is covered in Section 17.3. These are fundamental topics in web data management. We then discuss distributed XML data management (Section 17.4). Although the web pages were originally encoded using HTML, the use of XML and the prevalence of XML-encoded data are increasing, particularly in the data repositories available on the web. Therefore, the distributed management of XML data is increasingly important.

17.1 Web Graph Management

The web consists of "pages" that are connected by hyperlinks, and this structure can be modelled 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 nodes and the links between them are represented as directed edges [Kumar et al., 2000; Raghavan and Garcia-Molina, 2003; Kleinberg et al., 1999]. Studying the web graph is obviously of interest to theoretical computer scientists, because it exhibits a number of interesting characteristics, but it is also important for studying data management issues since the graph structure is exploited in web search [Kleinberg et al., 1999; Brin and Page, 1998; Kleinberg, 1999], categorization and classification of web content [Chakrabarti et al., 1998], and other web-related tasks. The important characteristics of the web graph are the following [Bonato, 2008]:

(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 is less than the number of vertices. This means that the each node of the graph has a limited number of neighbors, even if the nodes 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 consist 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 network." 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 network. The in- and out-degree distributions of the web graph follow power law distributions. This means that the probability that a node 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 [Broder et al., 2000].

This brings us to a discussion of the structure of the web graph, which has a "bowtie" shape (Figure 17.1) [Broder et al., 2000]. It has a strongly connected component (the knot in the middle) in which there is a path between each pair of pages. 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 exists 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 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.

A particularly relevant issue that needs to be addressed is the management of the very large, dynamic, and volatile web graph. In the remainder, we discuss two methods that have been proposed to deal with this issue. The first one compresses the web graph for more efficient storage and manipulation, while the second one suggests a special representation for the web graph.

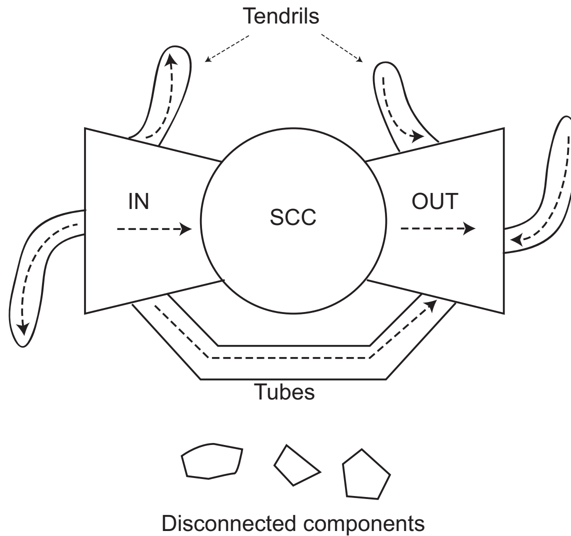

Fig. 17.1 The Structure of the web as a Bowtie (Based on [Kumar et al., 2000].)

17.1.1 Compressing Web Graphs

Compressing a large graph is well-studied, and a number of techniques have been proposed. However, the web graph structure is different from the graphs that are addressed by these techniques, which makes it difficult (if not impossible) to apply the well-known graph compression algorithms to web graphs. Thus, new approaches are needed.

A specific proposal for compressing the web graph takes advantage of the fact that we can attempt to find nodes that share several common *out-edges*, corresponding to the case where one node might have copied links from another node [Adler and Mitzenmacher, 2001]. The main idea behind this technique is that when a new node is added to the graph, it takes an existing page and copies some of the links from that page to itself. For example, a new page v might examine the out-edges from a page w and link to a subset of the pages that w links to. This intuition is based on the idea that the creator of a new page decides what pages to link to based on an existing page or pages that the page creator already likes [Kumar et al., 1999]. In this case, node w is called the *reference* for node v.

Given that the in-degree and out-degree of the web graph follow a Zipfian distribution, there is a large variance in the degrees. Thus, a Huffman-based compression scheme can be used. There are alternative compression methods in this class, but a simple one that demonstrates the idea is as follows.

Once the node from which links were copied has been identified, the difference between the out-edges of the two nodes can be identified. If node w is labelled as a reference of node v, a 0/1 bit vector can be generated that denotes which out-edges of w are also out-edges of node v. Other out-edges of v can be separately identified

using another bit vector. Then, the cost of compressing node v using node w as a reference can be expressed as follows:

$$Cost(v,w) = out_deg(w) + \lceil \log n \rceil * (|N(v) - N(w)| + 1)$$

where $N(v)$ and $N(w)$ represent the set of out-edges for nodes v and w, respectively, and n is the number of nodes in the graph. The first term identifies the cost of representing the out-edges of the reference node w, $\lceil \log n \rceil$ is the number of bits required to identify a node in a web graph with n nodes, and $(|N(v) - N(w)| + 1)$ represents the difference between the out-edges of the two nodes.

Given a description of a graph in this compressed format, let us consider how it could be determined where a link from node v encoded using node w as a reference actually points. If the corresponding link from node w is encoded using another node u as a reference, then it needs to be determined where the corresponding link from node u points. Eventually, a link is reached that is encoded without using a reference node (in order to satisfy this requirement, no cycles among references are allowed) at which point the search stops.

17.1.2 Storing Web Graphs as S-Nodes

An alternative to compressing the web graph is to develop special storage structures that allow efficient storage and querying. *S-Nodes* [Raghavan and Garcia-Molina, 2003] is one such structure that provides a two-level representation of the web graph. In this scheme, the web graph is represented by a set of smaller directed sub-graphs. Each of these smaller sub-graphs encodes the interconnections within a small subset of pages. A top-level directed graph, consisting of *supernodes* and *superedges* contains links to these smaller sub-graphs.

Given a web graph W_G, the S-Node representation can be constructed as follows. Let $P = N_1, N_2, ..., N_n$ be a partition on the vertex set of W_G. The following types of directed graphs can be defined (Figure 17.2):

Supernode graph: A supernode graph contains n vertices, one for each partition in P. In Figure 17.2, there is a supernode for each of the partitions N_1, N_2 and N_3. Supernodes are linked using superedges. A superedge E_{ij} is created from N_i to N_j if there is at least one page in N_i that points to some page in N_j.

Intranode graph: Each partition N_i is associated with an intranode graph $IntraNode_i$ that represents all the interconnections between the pages that belong to N_i. For example, in Figure 17.2, $IntraNode_1$ represents the hyperlinks between pages P_1 and P_2.

Positive superedge graph: A positive superedge graph $SEdgePos_{i,j}$ is a directed bipartite graph representing all links from N_i to N_j. In Figure 17.2, $SEdgePos_{1,2}$

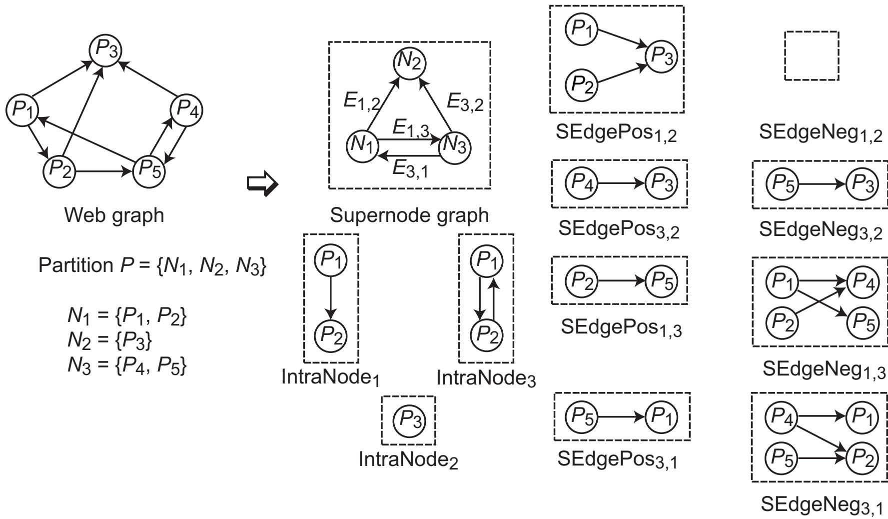

Fig. 17.2 Partitioning the web graph (Based on [Raghavan and Garcia-Molina, 2003].)

contains two edges that represent the two links from P_1 and P_2 to P_3. There is an $SEdgePos_{i,j}$ if there exists a corresponding superedge $E_{i,j}$.

Negative superedge graph: A negative superedge graph $SEdgeNeg_{i,j}$ is a directed bipartite graph that represents all links between N_i and N_j that do not exist in the actual web graph. Similar to $SEdgePos$, an $SEdgeNeg_{i,j}$ exists if and only if there exists a corresponding superedge $E_{i,j}$.

Given a partition P on the vertex set of W_G, an S-Node representation $SNode(W_G, P)$ can be constructed by using the supernode graph that points to the intranode graph and a set of positive and negative supernode graphs. The decision as to whether to use the positive or the negative supernode graph depends on which representation has the lower number of edges. Figure 17.3 shows the specific representation of an S-Node for the example given in Figure 17.2.

S-node representation exploits empirically observed properties of web graphs to guide the grouping of pages into super-nodes and uses compressed encodings for the lower level directed graphs. This compression allows the reduction of the number of bits needed to encode a hyperlink from 15 to 5 [Raghavan and Garcia-Molina, 2003], which in turn allows large web graphs to be loaded into main memory for processing. Furthermore, since the web graph is represented in terms of smaller directed graphs, it is possible to naturally isolate and locally explore portions of the web graph that are relevant to a particular query.

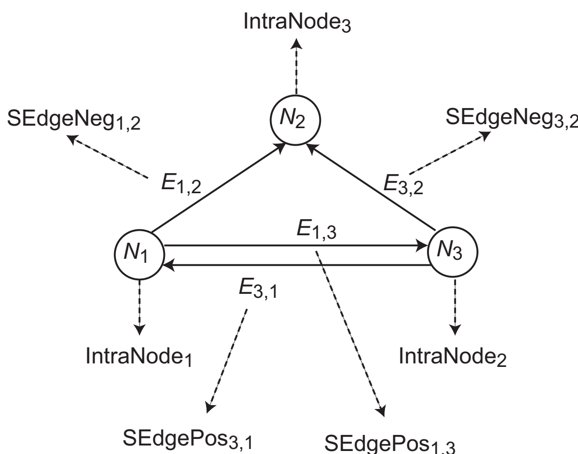

Fig. 17.3 S-node representation (Based on [Raghavan and Garcia-Molina, 2003].)

17.2 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 Figure17.4 [Arasu et al., 2001]. 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 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 17.2.1 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

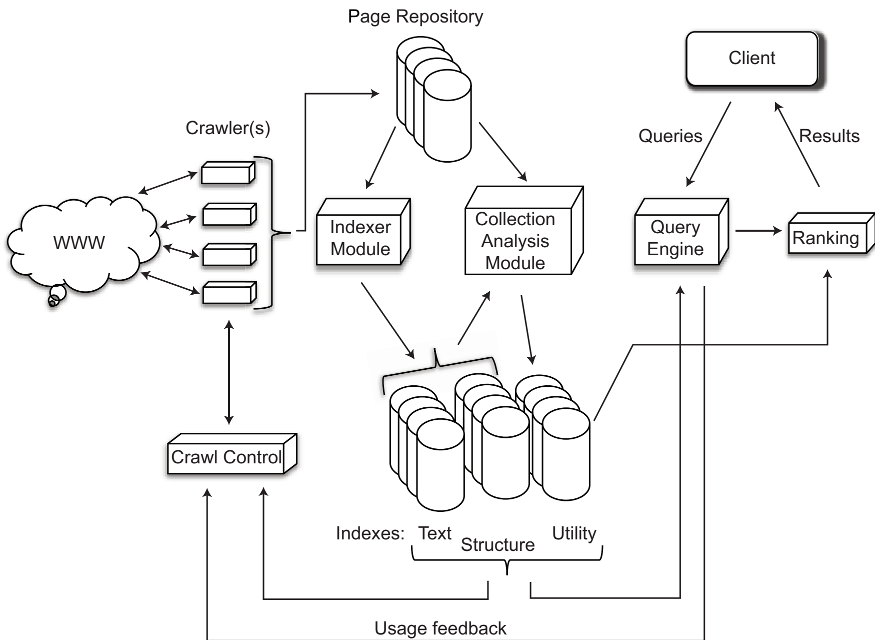

Fig. 17.4 Search Engine Architecture (Based on [**?**]

of pages. Section 17.2.2 explains current indexing technology and concentrates on ways indexes can be efficiently stored.

The *ranking module* is responsible for sorting the 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 17.2.3 introduces algorithms for ranking and describes approaches that exploit the link structure of the web to obtain improved ranking results.

17.2.1 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 [Cho et al., 1998]. 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 with respect to the number of pages that point to P (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 [Page et al., 1998] that is used by Google and others. A possible dynamic measure may be one that calculates the importance of a page P with respect its textual similarity to the query that is being evaluated using some of the well-known information retrieval similarity measures.

Let us briefly discuss the PageRank measure. The PageRank of a page P_i (denoted $r(P_i)$) is simply the normalized sum of the PageRank of all P_i's backlink pages (denoted as B_{P_i}):

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

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 links that P_j has to other pages. 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.

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* [Cho et al., 1998; Najork and Wiener, 2001]. 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 modelling 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 [Langville and Meyer, 2006]:

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

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.

In addition to the fundamental design issues discussed above, there are a number of additional concerns that need to be addressed for efficient implementation of crawlers. We discuss these briefly.

Since many web pages change over time, crawling is a continuous activity and pages need to be re-visited. Instead of restarting from scratch each time, it is preferable to selectively re-visit 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 re-visit 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 re-visited [Cho and Garcia-Molina, 2000]. 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 [Edwards et al., 2001]. *Sampling-based approaches* [Cho and Ntoulas, 2002] 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. They use learning techniques to identify the topic of a given page. Learning techniques are beyond our scope, but a number of them have been developed for this purpose, such as naïve Bayes classifier [Mitchell, 1997; Chakrabarti et al., 2002], and its extensions [Passerini et al., 2001; Altingövde and Ulusoy, 2004], reinforcement learning [McCallum et al., 1999; Kaelbling et al., 1996], 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* [Cho and Garcia-Molina, 2002].

17.2.2 Indexing

In order to efficiently search the crawled pages and the gathered information, a number of indexes are built as shown in Figure 17.4. The two more important indexes are the *structure* (or *link*) *index* and a *text* (or *content*) *index*. We discuss these in this section.

17.2.2.1 Structure Index

The structure index is based on the graph model that we discussed in Section 17.1, 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 Section 17.1. 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.

17.2.2.2 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* [Manber and Myers, 1990], *inverted files* or *inverted indexes* [Hersh, 2001], and *signature files* [Faloutsos and Christodoulakis, 1984]. 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 type of 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 the particular word occurs [Lim et al., 2003]. If needed, the location of the word in 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 [Brin and Page, 1998]. 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 $\langle B \rangle$ tags), in section headings (within $\langle H1 \rangle$ or $\langle H2 \rangle$ tags), or as anchor text might be weighted differently in the ranking algorithms [Arasu et al., 2001].

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 [Salton, 1989].

Constructing and maintaining an inverted index has three major difficulties that need to be addressed [Arasu et al., 2001]:

1. In general, building an inverted index involves processing each page, reading all 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 non-static collection like the web.

2. The rapid change of the web poses the second challenge for maintaining the "freshness" of the index. Although we argued in the previous section that incremental crawlers should be deployed to ensure freshness, it has also been argued that 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 [Melnik et al., 2001].
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 [Ribeiro-Neto and Barbosa, 1998]. We don't discuss these further, as the issues are similar to the distributed data and directory management issues we have already covered in previous chapters.

17.2.3 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 thus 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 expected to have good quality, and hence the number of incoming links to a page can be used as a ranking criteria. This intuition is the basis of ranking algorithms, but, of course, the each specific algorithm implements this intuition in a different and sophisticated way. We already discussed the PageRank algorithm earlier. We will discuss an alternative algorithm called HITS to highlight different ways of approaching the issue [Kleinberg, 1999].

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 non-negative 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.

17.2.4 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: dmoz (http://dmoz.org/), LookSmart (http://www.looksmart.com/), and Yahoo (http://www.yahoo.com/). The web directory is a hierarchical taxonomy that classifies human knowledge [Baeza-Yates and Ribeiro-Neto, 1999]. 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 server 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 Copernic (http://www.copernic.com/), Dogpile (http://www.dogpile.com/), MetaCrawler (http://www.metacrawler.com/), and Mamma (http://www.mamma.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.

17.3 Web Querying

Declarative querying and efficient execution of queries has 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.

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[2]. 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 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, a standardized language

[2] We are focusing on the "open" web here; deep web data may have a schema, but it is usually not accessible to users.

for XML has emerged (XQuery), and as XML becomes more prevalent on the web, this language is likely to become dominant and more widely used. We discuss XML data and its management in Section 17.4.

A number of different approaches to web querying have been developed, and we discuss them in this section.

17.3.1 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 [Buneman et al., 1996] are similar.

OEM (Object Exchange Model) [Papakonstantinou et al., 1995] 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 ⟨label, type, value, oid⟩, 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. One would immediately recognize the similarity between OEM object definition and the object models that we discussed in Chapter 15.

Example 17.1. 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 Figure 17.5. Each line shows one OEM object and the indentation is provided to simplify the display of the object structure. For example, the second line <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, &034); 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, the first book (&o2) has the price information that the second one (&o34) does not have, while the second one has ISBN and publisher information that the first does not have. The object-oriented structure of the database is obvious – complex objects consist of

```
<bib, complex, {&o2, &o22, &034}, &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, &o17}, &o7>
         <chapter, complex, {&o9, &o10}, &o8>
            <heading, string, "...", &o9>
            <body, string, "...", &o10>
            ...
         <chapter, complex, {&o18, &o19}, &17>
            <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,&o17,&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. 17.5 An Example OEM Specification

subobjects (books consist of chapters in addition to other information), and objects may be shared (e.g., `&o4` is shared by both `&o3` and `&o23`). ♦

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 node-labelled graph where the nodes correspond to each OEM object and the edges correspond to the subobject relationship. The label of a node is the oid and the label

of the object corresponding to that node. However, it is quite common in literature to model the data as an edge-labelled 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 node labels. In Example 17.2, we use a node and edge-labelled representation that shows oids as node labels and assigns edge labels as described above.

Example 17.2. Figure 17.6 depicts the node and edge-labelled graph representation of the example OEM database given in Example 17.1. Normally, each leaf node also contains the value of that object. To simplify exposition of the idea, we do not show the values. ♦

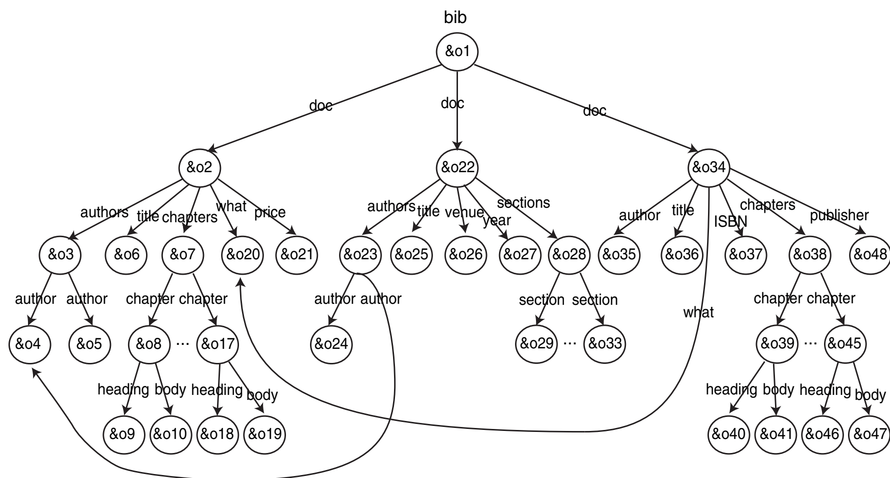

Fig. 17.6 The corresponding OEM graph for the OEM database of Example 17.1

The semistructured approach fits reasonably well for modelling web data that 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 Chapter 9.

Let us now focus on the languages that have been developed to query semistructured data. As noted above, we will focus our discussion by considering a particular language, Lorel [Papakonstantinou et al., 1995; Abiteboul et al., 1997], but other languages are similar in their basic approaches.

Lorel has changed over its development cycle, and the final version [Abiteboul et al., 1997] is defined as an extension of OQL discussed in Chapter 15. Thus, it has the familiar `SELECT-FROM-WHERE` structure, but path expressions can exist in the `SELECT`, `FROM` and `WHERE` clauses.

The fundamental construct in forming Lorel queries is, therefore, a *path expression*. We discussed path expressions as they appear in object database systems in Section 15.6.2.2, but we give the definition here as it applies to Lore. 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 labelled doc and then follow the edge labelled title. Note that there are three paths in Figure 17.6 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 are 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 17.3. 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 17.4. The following are example Lorel queries that use some of the path expressions given in Example 17.3:

(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
```

As can be observed, semistructured data approach to modelling 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 node is a simple entity) nor does it support ordering as there is no imposed ordering among the nodes 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 nodes. These cannot be separately modelled, 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 [Goldman and Widom, 1997] 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 light-weight schema, which is useful for browsing the database structure, formulating queries, storing statistical information, and enabling query optimization.

Example 17.5. The DataGuide corresponding to the OEM graph in Example 17.2 is given in Figure 17.7. ♦

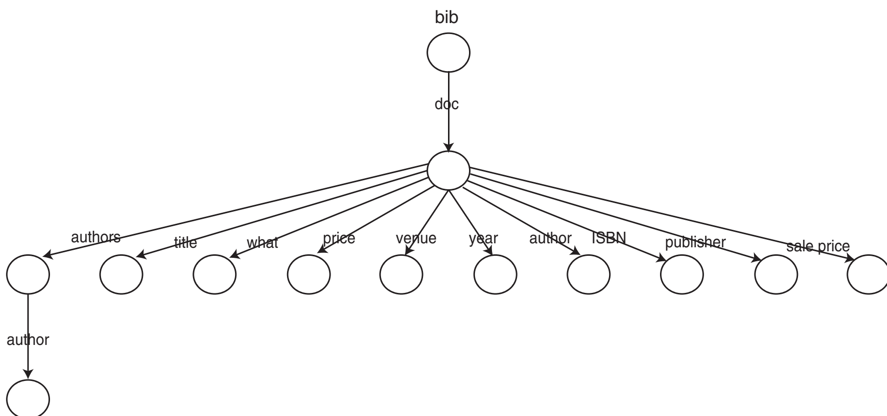

Fig. 17.7 The DataGuide corresponding to the OEM graph of Example 17.2

17.3.2 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) 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 [Florescu et al., 1998]. 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 [Mendelzon et al., 1997], W3QL [Konopnicki and Shmueli, 1995], and WebLog [Lakshmanan et al., 1996], while second generation approaches include WebOQL [Arocena and Mendelzon, 1998], and StruQL [Fernandez et al., 1997]. 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)

ANCHOR(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. ANCHOR 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 17.6.

Example 17.6. The following are examples of possible path expressions that can be specified in WebSQL [Mendelzon et al., 1997].

(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 [Mendelzon et al., 1997] with minor modifications) demonstrate the features of WebSQL.

Example 17.7. 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 aplets 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
```

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

As noted earlier, second generation languages, such as WebOQL, address this shortcoming by modelling 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-labelled 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 labelled 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 17.8. Let us revisit Example 17.1 and assume that instead of modelling 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 in Figure 17.8. 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 Figure 17.8, 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 (Figure 17.6), 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.

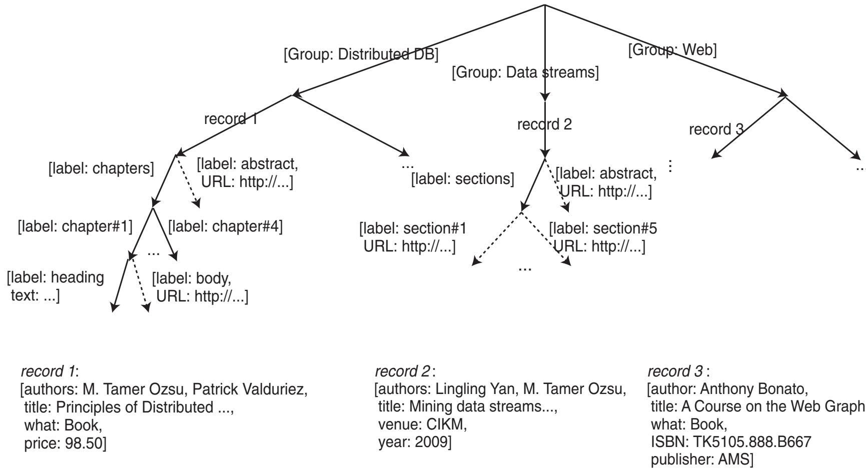

Fig. 17.8 The hypertree example

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

Prime: returns the first subtree of its argument (denoted ').

Peek: extracts a field from the record that labels the first outgoing edges of its document. This is the straightforward "dot notation" that we have seen multiple times before. For example, if x points to the root of the subtree reached from the "Groups = Distributed DB" edge, x.authors would retrieve "M. Tamer Ozsu, Patrick Valduriez".

Hang: builds an edge-labeled tree with a record formed with the arguments (denoted as []).

Example 17.9.
Let us assume that the tree depicted in Figure 17.9(a) is retrieved as a result of a query (call it Q1). Then the expression ["Label: "Papers by Ozsu" / Q1] results in the tree depicted in Figure 17.9(b). ♦

Concatenate: combines two trees (denoted +).

Example 17.10. Again, assuming that the tree depicted in Figure 17.9(a) is retrieved as a result of query Q1, Q1+Q2 produces tree in Figure 17.9(c). ♦

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.

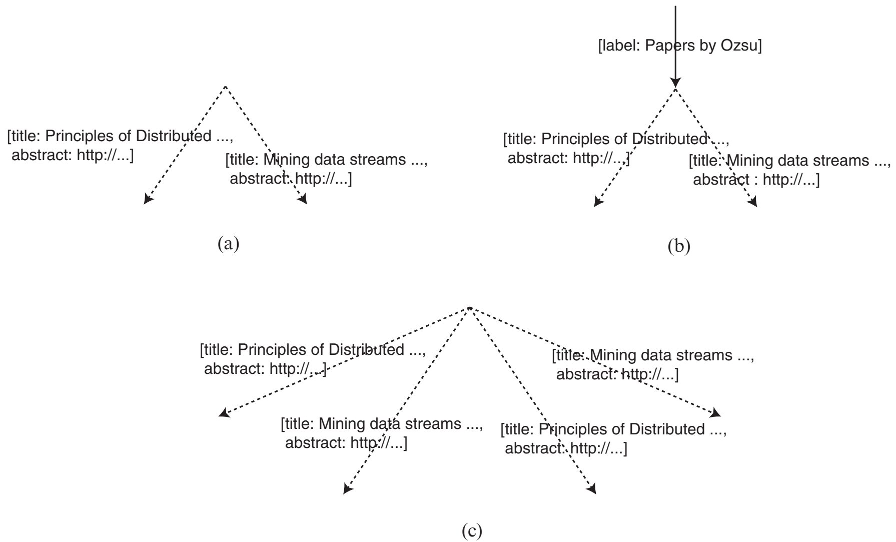

Fig. 17.9 Examples of Hang and Concatenate Operators

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 17.11. Let `dbDocuments` denote the documents in the database shown in Figure 17.8. Then the following query finds the titles and abstracts of all documents authored by "Ozsu" producing the result depicted in Figure 17.9(a).

```
SELECT [y.title, y'.URL]
FROM   x IN dbDocuments, y IN x'
WHERE  y.authors ~ "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 ∼), 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.

17.3.3 Question Answering

In this section, we discuss an interesting and unusual (from a database perspective) approach to querying 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 Figure 17.10. 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 [Kwok et al., 2001], WebQA [Lam and Özsu, 2002], Start [Katz and Lin, 2002], and Tritus [Agichtein et al., 2004]. There are also commercial systems with varying capabilities (e.g., Wolfram Alpha http://www.wolframalpha.com/).

We describe the general functionality of these systems using the reference architecture given in Figure 17.11. 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 run-time while answering the user questions. For example, Tritus employs a learning-based approach that uses a collection of frequently asked questions and their correct answers as a training data set. 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 question phrase). This is used to classify the question. In the second phase,

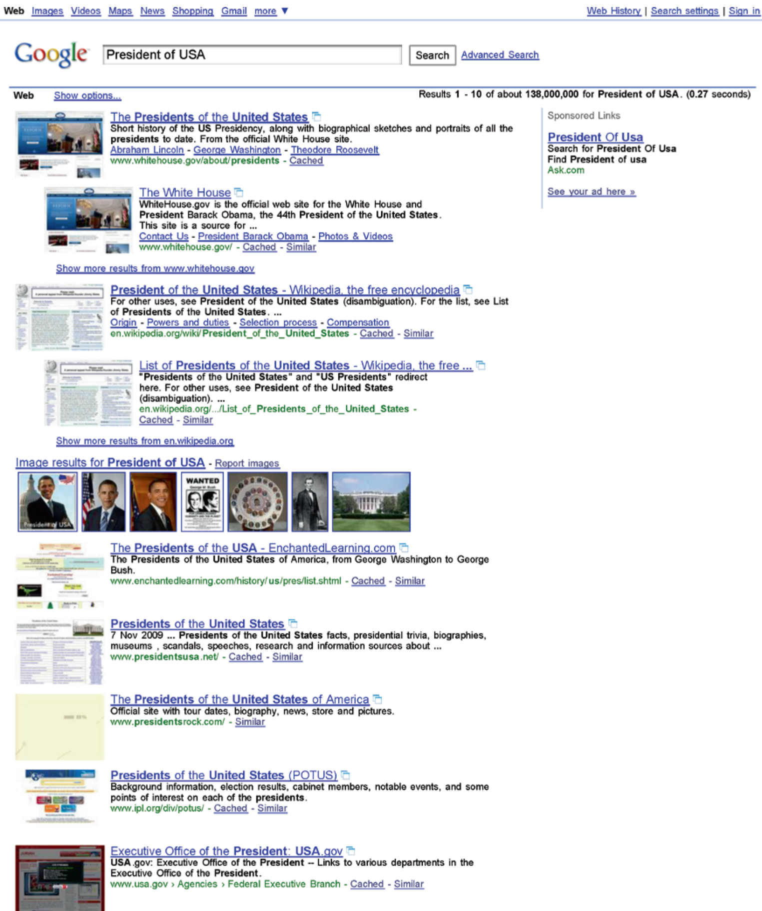

Fig. 17.10 Keyword Search Example

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 data set, 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 run-time 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

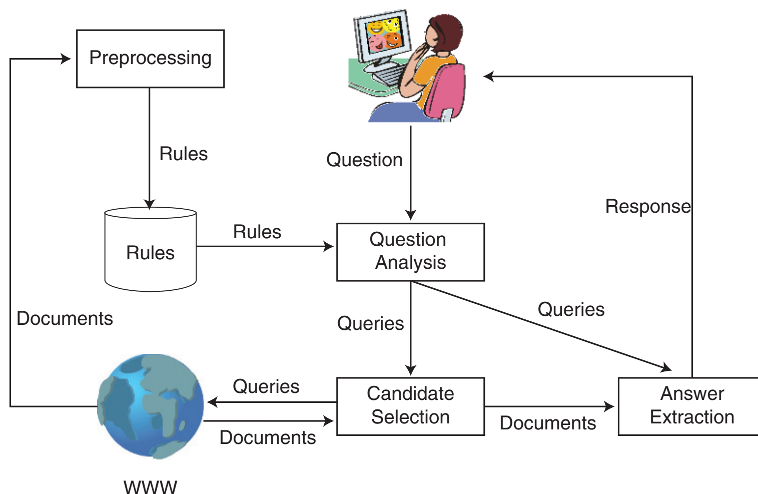

Fig. 17.11 General architecture of QA Systems

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 three classes: *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. It converts the user question into WebQAL, which is its internal language. The structure of WebQAL is

```
Category [-output Output-Option] -keywords Keyword-List
```

The user question is put in one of seven categories (Name, Place, Time, Quantity, Abbreviation, Weather, and Other). It generates a keyword list after stopword elimination and verb-to-noun conversion. Finally, it further refines the category information and determines the "output option", which is specific to each category. For example, given the question "Which country has the most population in the world?", WebQA would generate the WebQAL expression

```
Place -output country -keywords most population world
```

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 (e.g., WebQA uses the data sources extensively in addition to search engines). 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 what we will call "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 order[3]), 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 idiosynchrocies of natural language, and the difficulties of natural language processing.

17.3.4 Searching and Querying the Hidden Web

Currently, most general-purpose search engines only operate on the PIW while 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 searching 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 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 describe a number of research efforts that address these issues.

[3] The 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.

17.3.4.1 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.

Querying the Search Interface.

One approach is to analyze the search interface of the database, and build an internal representation for it [Raghavan and Garcia-Molina, 2001]. 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 [Lage et al., 2002]. 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).

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 [Lage et al., 2002]. 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 [Hedley et al., 2004b]. 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.

17.3.4.2 Metasearching

Metasearching is another approach for querying the hidden web. Given a user's query, a metasearcher performs the following tasks [Ipeirotis and Gravano, 2002]:

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.

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, $SampleDF(w)$ [Callan et al., 1999; Callan and Connell, 2001]. 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 [Callan and Connell, 2001].

An alternative is to use a focused probing technique that can actually classify the databases into a hierarchical categorization [Ipeirotis and Gravano, 2002]. 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 [Hedley et al., 2004a]. 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 pre-defined number of documents are retrieved, and then statistics are calculated based on the retrieved documents.

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 [Ipeirotis and Gravano, 2002]. 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 [Gravano et al., 1999] 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 [Ipeirotis and Gravano, 2002] 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.

17.4 Distributed XML Processing

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* (e.g., paragraph, heading) that are encapsulated by *tags* (e.g., <p> paragraph </p>). Increasingly, XML (which stands for Extensive Markup Language) [Bray et al., 2009] has emerged as the preferred encoding. Proposed as a simple syntax with flexibility, human-readability, and machine-readability in mind,

XML has been adopted as a standard representation language for data on the Web. Hundreds of XML schemata (e.g., XHTML [XHTML, 2002], DocBook [Walsh, 2006], and MPEG-7 [Martínez, 2004]) are defined to encode data into XML format for specific application domains. Implementing database functionalities over collections of XML documents greatly extends the power to manipulate these data.

In addition to be a data representation language, XML also plays an important role in data exchange between Web-based applications such as Web services. Web services are Web-based autonomous applications that use XML as a *lingua franca* to communicate. A Web service provider describes services using the Web Service Description Language (WSDL) [Christensen et al., 2001], registers services using the Universal Description, Discovery, and the Integration (UDDI) protocol [OASIS UDDI, 2002], and exchanges data with the service requesters using the Simple Object Access Protocol (SOAP) [Gudgin et al., 2007] (a typical workflow can be found in Figure 17.12). All these techniques (WSDL, UDDI, and SOAP) use XML to encode data. Database techniques are also beneficial in this scenario. For example, an XML database can be installed on a UDDI server to store all registered service descriptions. A high-level declarative XML query language, such as XPath [Berglund et al., 2007] or XQuery [Boag et al., 2007] (we will discuss these shortly), can be used to match specific patterns described by a service discovery request.

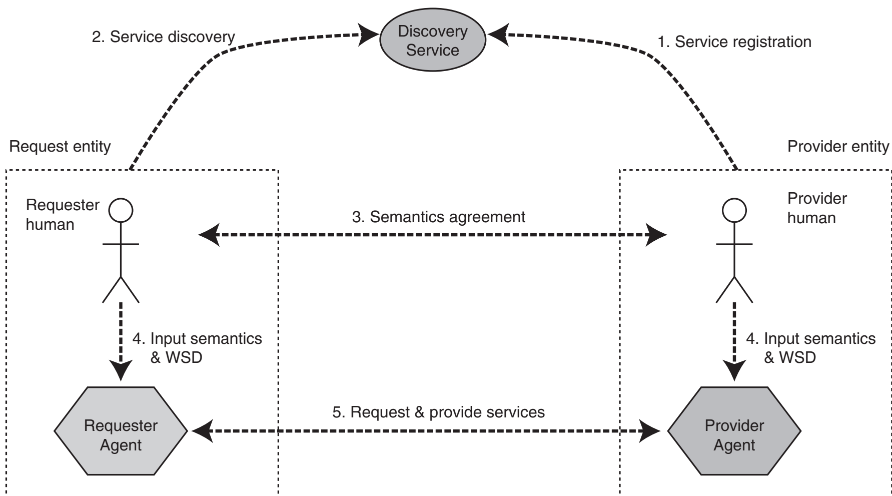

Fig. 17.12 A typical Web Service workflow suggested by the W3C Web Services Architecture (Based on [Booth et al., 2004].)

XML is also used to encode (or annotate) non-Web semistructured or unstructured data. Annotating unstructured data with semantic tags to facilitate queries has been studied in the text community for a long time (e.g., the OED project [Gonnet and Tompa, 1987]). In this scenario, the primary objective is not to share data with others

(although one can still do so), but to take advantage of the declarative query languages developed for XML to query the structure that is discovered through the annotation.

As noted above, XML is frequently used to exchange data among a wide variety of systems. Therefore, applications often access data from multiple, independently managed XML data collections. Consequently, a considerable amount of distributed XML processing work has focused on the use of XML in data integration scenarios. The major issues in this conext are similar to those that we have discussed in Chapters 4 and 9.

As the volume of XML data increases along with the workloads that operate on these data, efficient management of these collections become a serious concern. Similar to relational systems, centralized solutions are generally infeasible and distributed solutions are required. The issues here are analogous to the design of tightly-integrated distributed DBMSs that we have discussed in this book. However, the peculiarities of the XML data model and its query languages introduce important differences that we focus on in this section.

We start with a quick overview of XML and the two languages that have been defined for it: XPath and XQuery, particularly focusing on XPath since that has received more attention for its optimization (and since it is an important subset of XQuery). Then we summarize techniques for processing XML queries in a centralized setting as a prelude to the main part of the discussion, which focuses on fragmenting XML data, localizing XML queries by pruning unnecessary fragments, and, finally, their optimization. We should note that our objective is not to provide a complete overview of XML – the topic is much broader than can be covered in a section or a chapter, and there are very good sources, as we note at the end of this chapter, that treat the topic extensively.

17.4.1 Overview of XML

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, Figure 17.13 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 Figure 17.5 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` el-

```
<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. 17.13 An Example XML Document

ement). 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-IDREFs, 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[4].

An XML document is modelled 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 non-leaf nodes and is a strong 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 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 [Gao et al., 2009]. In this section, we will use a simpler schema definition that exploits the graph structure of XML documents as defined above [Kling et al., 2010].

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 $m : \Sigma \rightarrow \{\texttt{string}\}$. 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 $s(\psi) =$ ONCE, then an item of type σ_1 must contain exactly one item of σ_2. If $s(\psi) =$ OPT, then an item of type σ_1 may or may not contain an item of type σ_2. If $s(\psi) =$ 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.

[4] In addition, we omit the comment nodes, namespace nodes, and PI nodes from the XQuery Data Model.

Example 17.12. In the remainder of this chapter, we will use a slightly reorganized version of the XML example given in Figure 17.13. This is because that particular XML database consists of a single document, which is not suitable for demonstrating some of the distribution issues. The database definition can be modified by deleting the surrounding `<bib> </bib>` tags so that each book is one separate document in the database. However, we will make more changes to have an example that will better assist in the discussion of distribution issues. In this organization, the database will consist of multiple books, but organized by authors (i.e., the root of each document is an `<author>` element). This is given in Figure 17.14. ♦

Example 17.13. Let us revisit our bibliographic database and make a revision that the entries inside it are organized by authors rather than by publications and the only publications in the collection are books. In this case a (simplified) DTD definition is given below:

```
<?xml version="1.0"?>
<!DOCTYPE            author [
<!ELEMENT            author (name, pubs, agent?)
<!ELEMENT            pubs (book*)
<!ELEMENT            book (title,chapter*)
<!ELEMENT            chapter (reference?)
<!ELEMENT            reference (chapter)
<!ELEMENT            agent (name)
<!ELEMENT            name (first, last)
<!ELEMENT            first (CDATA)
<!ELEMENT            last (CDATA)
<!ATTLIST            book year CDATA #REQUIRED>
<!ATTLIST            book price CDATA #REQUIRED>
<!ATTLIST            author age CDATA #REQUIRED>
]
```

Instead of describing this DTD definition, we give its schema graph in Figure 17.15 using the notation introduced above, and this version clearly shows the semantics. Note that `CDATA` means that the content of the element is text. ♦

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 [Berglund et al., 2007] and XQuery [Boag et al., 2007] are two important query languages proposed by the World Wide Web Consortium (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 in the form of FLWOR expressions and we will briefly touch upon them when appropriate.

Although we have earlier defined path expressions, they take a particular form in the XPath context, so we will define them more carefully. A path expression consists of a list of *steps*, each of which consists of an *axis*, a *name test*, and zero or more *qualifiers*. The last step in the list is called a *return step*. There are in total thirteen

```
<author>
   <name>
      <first>M. Tamer </first>
      <last>Ozsu</last>
      <age>50</age>
   </name>
   <agent>
      <name>
         <first> John </first>
         <last> Doe </last>
      </name>
   </agent>
   <pubs>
      <book year = "1999", price = "$98.50">
         <title> Principles of Distributed ... </title>
         <chapter> ... </chapter>
         ...
         <chapter> ... </chapter>
      </book>
   </pubs>
</author>
<author>
   <name>
      <first>Patrick </first>
      <last>Valduriez</last>
      <age>40</age
   </name>
   <pubs>
      <book year = "1999", price = "$98.50">
         <title> Principles of Distributed ... </title>
         <chapter> ... </chapter>
         ...
         <chapter> ... </chapter>
      </book>
      <book year = "1992", price = "$50.00">
         <title> Data Management and Parallel Processing </title>
         <chapter> ... </chapter>
         ...
         <chapter> ... </chapter>
      </book>
   </pubs>
</author>
<author>
   <name>
      <first> Anthony </first>
      <last> Bonato </last>
      <age>30</age>
   </name>
   <pubs>
      <book year = "2008", price = "$75.00"
         <title> A Course on the Web Graph </title>
         <chapter> ... </chapter>
         ...
         <chapter> ... </chapter>
      </book>
   </pubs>
</author>
```

Fig. 17.14 A Different XML Document Example

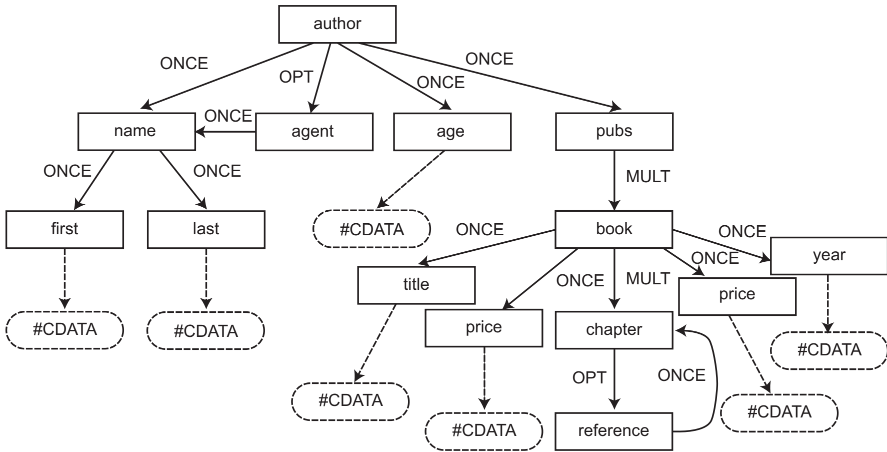

Fig. 17.15 Example XML Schema Graph for Fragmentation

axes, which are listed in Figure 17.16 together with their abbreviations if any. A name test filters nodes by their element or attribute names. Qualifiers are filters testing more complex conditions. The brackets-enclosed expression (which is usually called a *branching predicate*) can be another path expression or a comparison between a path expression and an atomic value (which is a string). The syntax of path expression is as follows:

```
     Path ::= Step (“/”Step)*
     Step ::= axis“ :: ”NameTest (Qualifier)*
 NameTest ::= ElementName | AttributeName | “∗”
Qualifier ::= ‘‘[”Expr“]”
     Expr ::= Path (Comp Atomic)?
     Comp ::= “ = ”|“ > ”|“ < ”|“ >= ”|“ <= ”|“! = ”
   Atomic ::= “′”String“′”
```

While the path expression defined here is a fragment of the one defined in XQuery [Boag et al., 2007] (by omitting features related to comments, namespaces, PIs, IDs, and IDREFs, as noted earlier), this definition still covers a significant subset and can express complex queries. As an example, the path expression

```
/author[.//last = "Valduriez"]//book[price < 100]
```

finds all books written by Valduriez with the book price less than 100.

As seen from the above definition, path expressions have three types of constraints: the *tag name constraints*, the *structural relationship constraints*, and the *value constraints*. The tag name, structural relationship, and value constraints correspond to the name tests, axes, and value comparisons in the path expression, respectively. A

Axes	Abbreviations
child	/
descendant	
descendant-or-self	//
parent	
attribute	/@
self	.
ancestor	
ancestor-or-self	
following-sibling	
following	
preceding-sibling	
preceding	
namespace	

Fig. 17.16 Thirteen axes and their abbreviations

path expression can be modeled as a tree, called a *query tree pattern* (QTP) $G(V,E)$ as follows (where V and E are sets of vertices and edges, respectively):

- each step is mapped to an edge in E;
- a special root node is defined as the parent of the tree node corresponding to the first step;
- if one step s_i immediately follows another step s_j, then the node corresponding to s_i is a child of the node corresponding to s_j;
- if step s_i is the first step in the branching predicate of step s_j, then the node corresponding to s_i is a child of the node corresponding to s_j;
- if two nodes represent a parent-child relationship, then the edge in E between them is labeled with the axis between their corresponding steps;
- the node corresponding to the return step is marked as the return node;
- if a branching predicate has a value comparison, then the node corresponding to the last step of the branching predicate is associated with an atomic value and a comparison operator.

For example, the QTP of the path expression

```
/author[last = "Valduriez"]//book[price < 100]
```

is shown in Figure 17.17. In this figure, the node `root` is the root node and the return node (`book`) is identified by two concentric ellipses.

While path expression is an important language component in XQuery, it is only one component of the XQuery language. A major language construct in XQuery is FLWOR expression, which consists of “for”, “let”, “where”, “order by” and “return” clauses. Each clause can reference path expressions or other FLWOR expressions recursively. A FLWOR expression iteration over a list of XML nodes, to bind a list

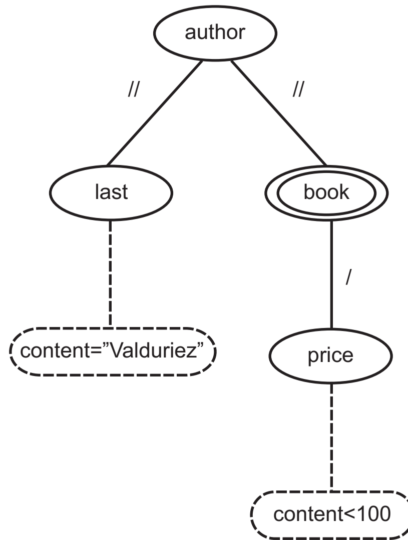

Fig. 17.17 A QTP of expression `/author[.//last = "Valduriez"]//book[price < 100]`

of nodes to a variable, to filter a list of nodes based on predicates, to sort the results, and to construct a complex result structure.

In essence, FLWOR is similar to the select-from-where-orderby statement found in SQL, except that the latter operates on a set or bag of tuples while the former manipulates a list of XML document tree nodes. Due to this similarity, FLWOR expressions may be rewritten into SQL statements leveraging existing SQL engines [Liu et al., 2008]. Another approach is to evaluate XQuery using a *native* evaluation engine [Fernández et al., 2003; Brantner et al., 2005]. We will discuss these approaches in the next section.

Example 17.14. The following FLWOR expression returns a list of books with its title and price ordered by their authors names (assuming the database, i.e., the XML document collection, is called "bib").

```
let $col := collection("bib")
for $author in $col/author
   order by $author/name
   for $b in $author/pubs/book
      let $title := $b/title
      let $price := $b/price
      return $title, $price
```

17.4.2 XML Query Processing Techniques

In this section we summarize some of the XML query processing techniques. Again, our objective is not to give an exhaustive coverage of the topic, since that would require an entire book in itself, but only to highlight the major issues.

There are three basic approaches to storing XML documents in a DBMS [Zhang and Özsu, 2010]: (1) the large object (LOB) approach that stores the original XML documents as-is in a LOB column (e.g., [Krishnaprasad et al., 2005; Pal et al., 2005]), (2) the extended relational approach that shreds XML documents into object-relational (OR) tables and columns (e.g., [Zhang et al., 2001; Boncz et al., 2006]), and (3) the native approach that uses a tree-structured data model, and introduces operators that are optimized for tree navigation, insertion, deletion and update (e.g., [Fiebig et al., 2002; Nicola and der Linden, 2005; Zhang et al., 2004]). Each approach has its own advantages and disadvantages.

The LOB approach is very similar to storing the XML documents in a file system, in that there is minimum transformation from the original format to the storage format. It is the simplest one to implement and support. It provides byte-level fidelity (e.g., it preserves extra white spaces that may be ignored by the OR and the native formats) that could be needed for some digital signature schemes. The LOB approach is also efficient for inserting (extracting) the whole documents to (from) the database. However it is slow in processing queries due to unavoidable XML parsing at query execution time.

In the extended relational approach, XML documents are converted to object-relational tables, which are stored in relational databases or in object repositories. This approach can be further divided into two categories based on whether or not the XML-to-relational mapping relies on XML Schema. The OR storage format, if designed and mapped correctly, could perform very well in query processing, thanks to many years of research and development in object-relational database systems. However, insertion, fragment extraction, structural update, and document reconstruction require considerable processing in this approach. For schema-based OR storage, applications need to have a well-structured, rigid XML schema whose relational mapping is tuned by a database administrator in order to take advantage of this storage model. Loosely structured schemas could lead to unmanageable number of tables and joins. Also, applications requiring schema flexibility and schema evolution are limited by those offered by relational tables and columns. The result is that applications encounter a large gap: if they cannot map well to an object-relational way of life due to tradeoffs mentioned above, they suffer a big drop in performance or capabilities.

Native XML storage approach stores XML documents using special data structures and formats that are designed for XML data. There is not, and should not be, a single native format for storing XML documents. Native XML storage techniques treat XML document trees as first class citizens and develop special purpose storage schemes without relying on the existence of an underlying database system. Since it is designed specifically for XML data model, native XML storage usually provides well-balanced tradeoffs among many criteria. Some storage formats may be designed to focus on

one set of criteria, while other formats may emphasize another set. For example, some storage schemes are more amenable to fast navigation, and some schemes perform better in fragment extraction and document reconstruction. Therefore, based on their own requirements, different applications adopt different storage schemes to trade off one set of features over another. As an example, Natix [Kanne and Moerkotte, 2000] partitions large XML document trees into small subtrees each of which can fit into a disk page. Inserting a node usually only affects the subtree in which the node is inserted. However, native storage systems may not be efficient in answering certain types of queries (e.g., `/author//book//chapter`) since they require at least one scan of the whole tree. The extended relational storage, on the other hand, may be more efficient for this query due to the special properties of the node encodings. Therefore, a storage system that balances the evaluation and update costs still remains a challenge.

Processing of path queries can also be classified into two categories: join-based approach [Zhang et al., 2001; Al-Khalifa et al., 2002; Bruno et al., 2002; Gottlob et al., 2005; Grust et al., 2003] and navigational approach [Barton et al., 2003; Josifovski et al., 2005; Koch, 2003; Brantner et al., 2005]. Storage systems and query processing techniques are closely related in that the join-based processing techniques are usually based on extended relational storage systems and the navigational approach is based on native storage systems. All techniques in the join-based approach are based on the same idea: each location step in the expression is associated with an input list of elements whose names match with the name test of the step. Two lists of adjacent location steps are joined based on their structural relationships. The differences between different techniques are in their join algorithms, which take into account the special properties of the relational encoding of XML document trees.

The navigational processing techniques, built on top of the native storage systems, match the QTP by traversing the XML document tree. Some navigational techniques (e.g., [Brantner et al., 2005]) are query-driven in that each location step in the path expressions is translated into an algebraic operator which performs the navigation. A data-driven navigational approach (e.g., [Barton et al., 2003; Josifovski et al., 2005; Koch, 2003]) builds an automaton for a path expression and executes the automaton by navigating the XML document tree. Techniques in the data-driven approach guarantee worst case I/O complexity: depending on the expressiveness of the query that can be handled, some techniques (e.g., [Barton et al., 2003; Josifovski et al., 2005]) require only one scan of the data, and the others (e.g., [Koch, 2003]) require two scans.

Both the join-based and navigational approaches have advantages and disadvantages. The join-based approach, while efficient in evaluating expressions having descendent-axes, may not be as efficient as the navigational approach in answering expressions only having child-axes. A specific example is `/*/*`, where all children of the root are returned. As mentioned earlier, each name test (`*`) is associated with an input list, both of which contain all nodes in the XML document (since all element names match with a wildcard). Therefore, the I/O cost of the join-based approach is $2n$, where n is the number of elements. This cost is much higher than the cost of the navigational operator, which only traverses the root and its children. On the other

hand, the navigational approach may not be as efficient as the join-based approach for a query such as `/author//book//chapter`, since the join-based approach only needs to read those elements whose names are `book` or `chapter` and join the two lists, but the navigational approach needs to traverse all elements in the tree. Therefore, a technique that combines the best of both approaches would be preferable.

As in relational databases, query processing is significantly aided by the existence of indexes. XML indexing approaches can be categorized into three groups. Some of the indexing techniques are proposed to expedite the execution of existing join-based or navigational approaches (e.g., XB-tree [Bruno et al., 2002] and XR-tree Jiang et al. [2003] for the holistic twig joins). Since these are special-purpose indexes that are designed for a particular baseline operator, their application is quite limited. Another line of research focuses on string-based indexes (e.g., [Wang et al., 2003b; Zezula et al., 2003; Rao and Moon, 2004; Wang and Meng, 2005]). The basic idea is to convert the XML document trees as well as the QTPs into strings and reduce the tree pattern matching problem to string pattern matching. Still other XML indexing techniques focus on the structural similarity of XML document tree nodes and group them accordingly [Milo and Suciu, 1999; Goldman and Widom, 1997; Kaushik et al., 2002]. Although different indexes may be based on different notions of similarity, they are all based on the same idea: similar tree nodes are clustered into equivalence classes (or *index nodes*), which are connected to form a tree or graph. FIX [Zhang et al., 2006b] follows a different approach and indexes the numerical features of subtrees in the data. Features are used as the index keys to a mature index such as B^+-tree. For each incoming query, the features of the query tree are extracted and used as search keys to retrieve the candidate results.

Finally, as we noted a number of times in earlier chapters, a cost-based optimizer is crucial to choosing the "best" query plan. The accuracy of cost estimation is usually dependent on the cardinality estimation. Cardinality estimation techniques for path expressions first summarize an XML document tree (corresponding to a document) into a small synopsis that contains structural information and statistics. The synopsis is usually stored in the database catalog and is used as the basis for estimating cardinality. Depending on how much information is reserved, different synopses cover different types of queries. DataGuide, that we introduced earlier, is on example. Recall that it records all distinct paths from a data set and compresses them into a compact graph. Path tree [Aboulnaga et al., 2001] is another example that follows the same approach (i.e., capturing all distinct paths) and is specifically designed for XML document trees. Path trees can be further compressed if the resulting synopsis is too large. Markov tables [Aboulnaga et al., 2001], on the other hand, do not capture the full paths but sub-paths under a certain length limit. Selectivity of longer paths are calculated using fragments of sub-paths similar to the Markov process. These synopsis structures only support simple linear path queries that may or may not contain descendent-axes. Structural similarity-based synopsis techniques (XSketch [Polyzotis and Garofalakis, 2002] and TreeSketch [Polyzotis et al., 2004]) are proposed to support branching path queries (i.e., those that contain branching predicates as defined earlier). These techniques are very similar to the

structural similarity-based indexing techniques: clustering structurally similar nodes into equivalence classes. An extra step is needed for the synopsis: summarize the similarity graph under some memory budget. A common problem of these heuristics is that the synopsis construction (expansion or summarization) time is still prohibitive for structure-rich data. XSEED [Zhang et al., 2006a] also follows the structural similarity approach and constructs a synopsis by first compressing an XML document to a small kernel, and then adds more information to the synopsis to improve accuracy. The amount of additional information is controlled by the memory availability.

Let us now consider XQuery FLWOR expression and introduce possible techniques for its evaluation. As mentioned in the previous subsection, one way to execute FLWOR expressions is to translate them into SQL statements, which can then be evaluated using existing SQL engines. One barrier however is that FLWOR expression works on the XML data model (list of XML nodes) but SQL takes relations as input. The translation has to introduce new operators or functions to convert data between these two data models. One major syntactic construct of this conversion is through the XMLTable function found in SQL/XML [Eisenberg et al., 2008]. XMLTable takes an XML input data source, an XQuery expression to generate rows, and outputs a list of rows with columns specified by the function as well.

Example 17.15. As an example, the following XMLTable function

```
XMLTable('/author/name'
passing collection('bib')
columns
first varchar2(200) PATH '/name/first',
last varchar2(200) PATH '/name/last')
```

takes the input document `bib.xml` from the "passing" clause and applies the path expression `/bib/book` to the input document. For each matching book, there will be one row generated. For each row there are two columns specified by the "columns" clause with its column name and type. A path expression is also given to each column to be used to evaluate its value. The semantics of this XMLTable function is the same as the FLWOR expression:

```
for $a in collection('bib')/author/name
return {$a/first, $a/last}
```

In fact, almost all FLWOR expressions can be translated to SQL with the help of the XMLTable function. Therefore, the XMLTable function maps XQuery results to relational tables. The result of XMLTable can then be treated as a virtual table and any other SQL construct can be composed on top of that.

Another approach to evaluating XQuery statements is to implement a native XQuery engine that interprets XQuery statements on top of XML data. One example is Galax [Fernández et al., 2003] that first takes an XQuery expression and normalizes it into XQuery core [Draper et al., 2007], which is a covering subset of XQuery. The XQuery core expression is then statically type-checked against the XMLSchema associated with the input data. When the input XML data are parsed and the instance

of XML data model (DOM) is generated, the XQuery core expression is dynamically evaluated on the instance of the data model.

Natix [Brantner et al., 2005] is another native approach, but one that defines a set of algebraic operators to which XPath or XQuery queries can be translated. Similar to the relational system, optimization rules can be applied to the operator tree to rewrite it into a more efficient plan. Moreover, Natix defines a native XML storage format based on tree partitioning. Large XML document trees can be partitioned into smaller ones, each of which can fit into a disk page. This native storage format is more scalable than main memory-based DOM representation, and it allows more efficient tree navigation and potentially more efficient path expression evaluation.

In addition to pure relational and pure native XQuery evaluation techniques, there are others that follow a hybrid approach. For example, MonetDB/XQuery [Boncz et al., 2006] stores XML data as a relational table based on the nodes' pre- and post-order position when traversing the tree. XQuery statements are translated into physical relational operators that are designed for efficient evaluation. One particular example is the staircase join operator designed for efficient evaluation of path expressions. In this way, it relies on the SQL engine for most of the relational operations, and expedites XML-specific tree navigations by special purpose operators. In fact, many commercial database vendors also implement special operators in their relational SQL engine to speed up path expression evaluation (e.g., Oracle [Zhang et al., 2009a]). Therefore, while many XQuery engines leverage SQL engines for their ability to efficiently evaluate SQL-like functionalities, many XML specific optimizations and implementations now also penetrate into SQL engine implementations.

17.4.3 Fragmenting XML Data

If we follow the distribution methodology that we introduced earlier in the book, the first step is fragmentation and distribution of data to various sites. In this context, a relevant question is what it means to fragment XML data, and whether we can define horizontal and vertical fragmentation analogous to relational systems. As we will see, this is possible.

Let us first take a detour and consider an interesting case that we refer to as *ad hoc fragmentation*. In this case, there is no explicit, schema-based fragmentation specification; XML data are fragmented by arbitrarily cutting edges in XML document graphs. One example that follows this approach is Active XML [Abiteboul et al., 2008a], which represents cross-fragment edges as calls to remote functions. When such a function call is activated, the data corresponding to the remote fragment are retrieved and made available for local processing. An active XML document, therefore, consists of a static part, which is the XML data, and a dynamic part that includes the function call to web services. When this document is accessed and the service call is invoked, the returned data (i.e., a data fragment) is inserted in place of the call. Although originally designed for easy service integration by allowing calls to various web services, active XML inherently exploits the distribution of data. One

way to view this approach is that data fragments are shipped from the source sites to where the XML document is located. When the required data are gathered at this site, and the query is executed on the resulting document.

Example 17.16. Consider the following active XML document where a function call (getPubs) is embedded into a static XML document:

```
<author>
   <name>
      <first> J. </first>
      <last> Doe </last>
   </name>
   ...
   <call fun="getPubs('J. Doe')" />
</author>
```

The resulting document, following the invocation of the function call, would be as follows:

```
<author>
   <name>
      <first> J. </first>
      <last> Doe </last>
   </name>
   ...
   <pubs>
         <book> ... </book>
         ...
   </pubs>
</author>
```

Ad hoc fragmentation works well when the data are already distributed. However, extending it the case where an XML data graph is partitioned arbitrarily is problematic, since it may not be possible to specify the fragmentation predicate clearly. This would decrease the opportunities for distributed query optimization. Remember that distributed optimization in the relational context heavily depends upon the existence of a precise definition of the fragmentation predicate.

The alternative that addresses this issue is *structure-based fragmentation*, which is based on the concept of fragmenting an XML data collection based on some properties of the schema. This is analogous to what we have discussed in the relational setting. The first issue that arises is what types of fragmentations we can define. Similar to relational systems, we can distinguish between horizontal fragmentation where subsets of the data are selected, and vertical fragmentation where fragments are identified based on "projections" over the schema. The specific definitions of these differ among various works; we will follow one line of research to illustrate the concepts [Kling et al., 2010].

A horizontal fragmentation can be defined by a set of fragmentation predicates, such that each fragment consists of the document trees that match the corresponding predicate. For a horizontal fragmentation to be meaningful, the data should consist

of multiple document trees; otherwise it makes no sense to have fragments such that each fragment follows the same schema, which is a requirement of horizontal fragmentation. These document trees can either be entire XML documents or they can be the result of a previous vertical fragmentation step. Let $D = \{d_1, d_2, \ldots, d_n\}$ be a collection of document trees such that each $d_i \in D$ follows to the same schema. Then we can define a set of *horizontal fragmentation predicates* $P = \{p_0, p_1, \ldots, p_{l-1}\}$ such that $\forall d \in D : \exists$ unique $p_i \in P$ where $p_i(d)$. If this holds, then $F = \{\{d \in D \mid p_i(d)\} \mid p_i \in P\}$ is a set of horizontal fragments corresponding to collection D and predicates P.

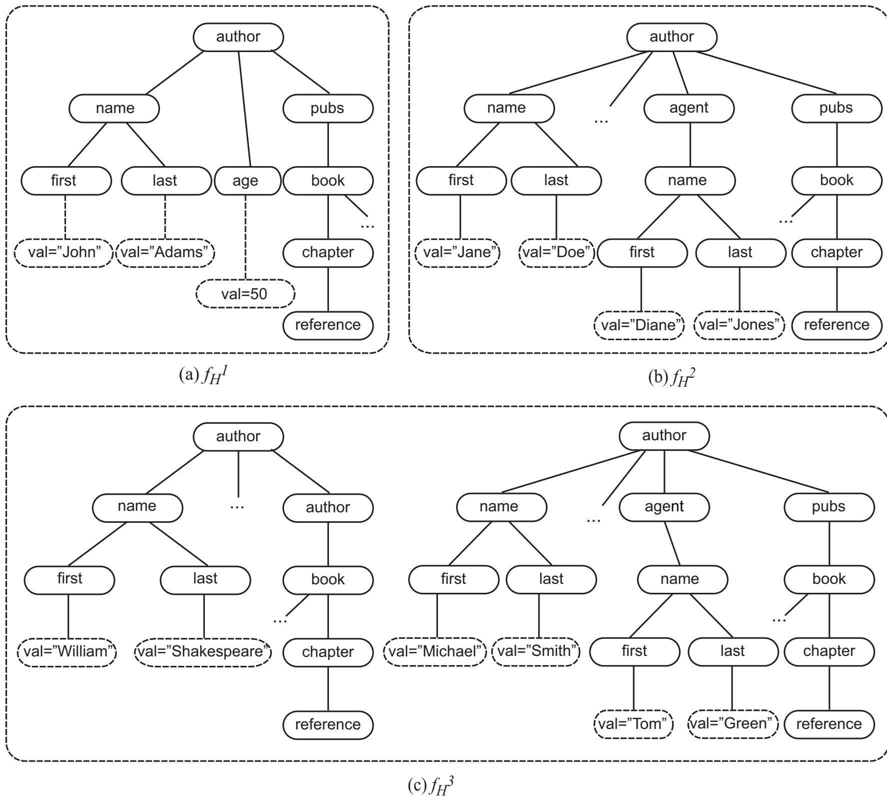

Fig. 17.18 Horizontally Fragmented XML Database

Example 17.17. Consider a bibliographic database that conforms to the schema given in Example 17.13 (and Figure 17.15). A possible horizontal fragmentation of this database based on the first letter of authors' last names is given in Figure 17.18. In this case, we are assuming that there are only four authors in the database whose names are "John Adams", "Jane Doe", "Michael Smith", and "William Shakespeare".

Note that we do not show all of the attributes of elements; in particular, the age attribute of authors, and the price attribute of books are not always shown.

If we assume that, in the example schema, m(`last`) is the set of strings that start with upper-case letters of the English alphabet, then the fragmentation predicates are straightforward. Note that the fragmentation predicates can be represented as trees referred to as *fragmentation tree patterns* (FTPs) [Kling et al., 2010] shown in Figure 17.19 where the edges are labelled with the corresponding XPath axis. ♦

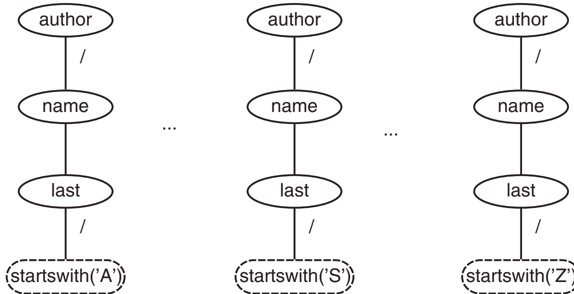

Fig. 17.19 Example Fragmentation Tree Patterns

Definition of vertical fragmentation is more interesting. A vertical fragmentation is defined by fragmenting the schema graph of the collection into disjoint subgraphs. Formally, given a schema as defined earlier, we can define a *vertical fragmentation function* $\phi : \Sigma \rightarrow F_\Sigma$ where F_Σ is a partitioning of Σ (recall that Σ is the set of node types). The fragment that has the root element is called the *root fragment*; the concepts of *parent fragment* and *child fragment* can be defined in a straightforward manner.

Example 17.18. Figure 17.20 shows a fragmented schema graph that corresponds to the schema that we have been considering. The item types have been fragmented into four disjoint subgraphs. Fragment f_V^1 consists of the item types `author` and `agent`, fragment f_V^2 consists of the item types `name`, `first` and `last` along with their text content, fragment f_V^3 consists of `pubs` and `book` and fragment f_V^4 includes the item types `chapter` and `reference`.

The vertical fragment instances of our example database are given in Figure 17.21, where f_V^1 is the root fragment. Again, we do not show all the nodes and we have omitted "val=" from the value nodes to fit the figure (these are done in Figure 17.22 as well). ♦

As depicted in Figure 17.21, there are document edges that cross fragment boundaries. To facilitate these connections, special nodes are introduced in the fragments: for an edge from fragment f_i to f_j, a *proxy node* is introduced in the originating fragment f_i (denoted $P_k^{i \rightarrow j}$ where k is the ID of the proxy node) and a *root proxy node* is introduced in the target fragment f_j (denoted $RP_k^{i \rightarrow j}$). Since $P_k^{i \rightarrow j}$ and $RP_k^{i \rightarrow j}$

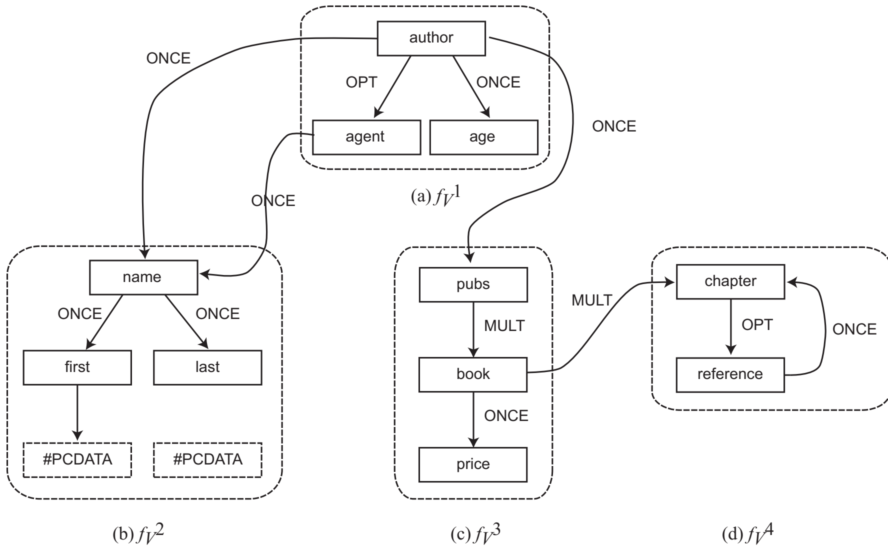

Fig. 17.20 Example Vertical Fragmentation of Schema

share the same ID (k) and reference the same fragments ($i \rightarrow j$), they correspond to each other and together represent a single cross-fragment edge in the collection.

Example 17.19. Figure 17.22 depicts the same fragmentation shown in Figure 17.21 with the proxy nodes inserted. ♦

Vertical fragments generally consist of multiple unconnected pieces of XML data if the database consists of multiple documents. In this case, each piece comes from one document, and can be referred to as a *document snippet*. In Figure 17.21 (and in Figure 17.22), fragment f_V^1 contains four snippets, each of which consists of the `author` and `agent` nodes of one of the documents in the database.

Based on the above definitions, fragmentation algorithms can be developed. This area is still not fully developed, therefore we will provide a general discussion rather than giving detailed algorithms.

The horizontal fragmentation algorithm for relational systems that we introduced in Chapter 3 can be used for XML databases as well with the appropriate revisions. Recall that the relational fragmentation algorithm is based on minterm predicates, which are conjunctions of simple predicates on individual attributes. Thus, the issue is how to transform the predicates found in QTPs (i.e., trees that correspond to queries) into simple predicates. There may be multiple ways of doing this. Kling et al. [2010] discuss one approach where the mapping is straightforward if the QTP does not contain descendent (//) axes; if they do, then these are "unrolled" into equivalent paths comprised entirely of child axes using schema information.

In the case of vertical fragmentation, the problem is somewhat more complicated. One way to formalize the problem is to use a cost model to estimate the response

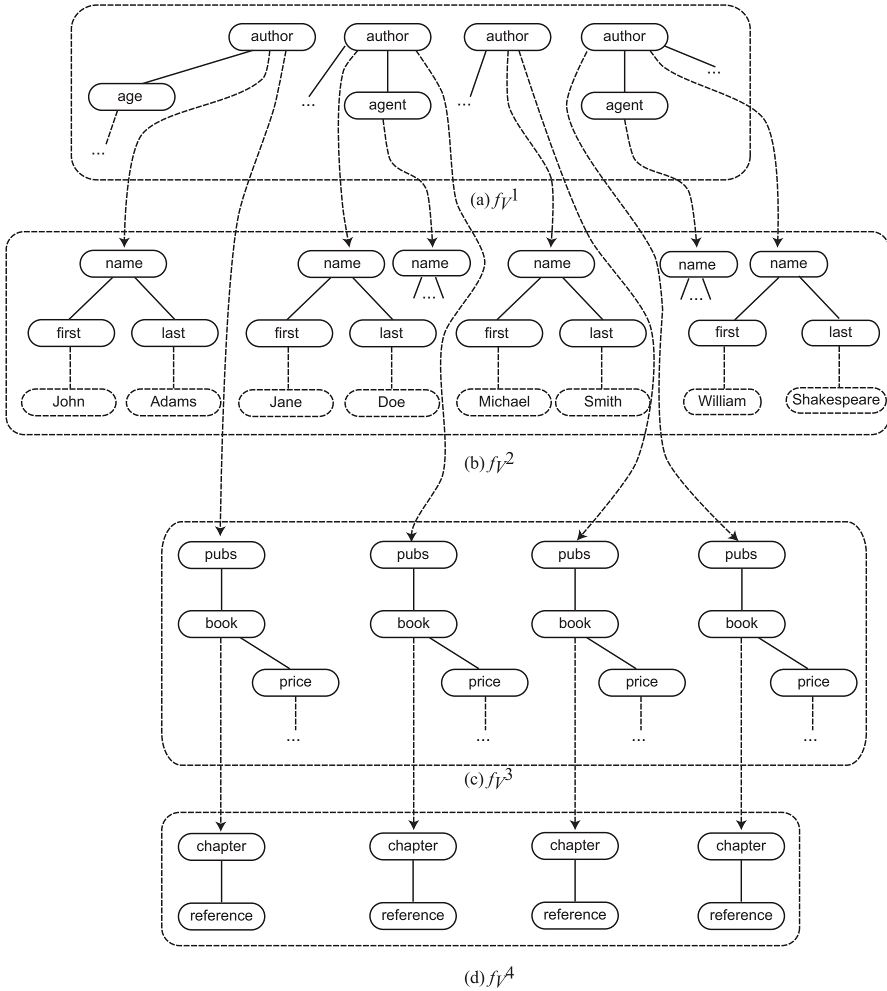

Fig. 17.21 Example Vertical Fragmentation Instances

time of the local query plans corresponding to each fragment. Since these local query plans are evaluated independently of each other in parallel, we can model the overall cost of a query as the maximum local plan cost. In theory, we can then enumerate all possible ways of partitioning the schema. Unfortunately, the large number of partitions to consider makes this approach infeasible for all but the smallest schemas. For a schema with n node types there are B_n partitions to consider where B_n is the nth Bell number, which is exponential in n (this is similar to the relational case). It is, however, possible to use a greedy strategy and still obtain a good fragmentation schema: Starting with a fragmentation schema in which each node type is placed in its own fragment, one can repeatedly merge the fragment corresponding to the most

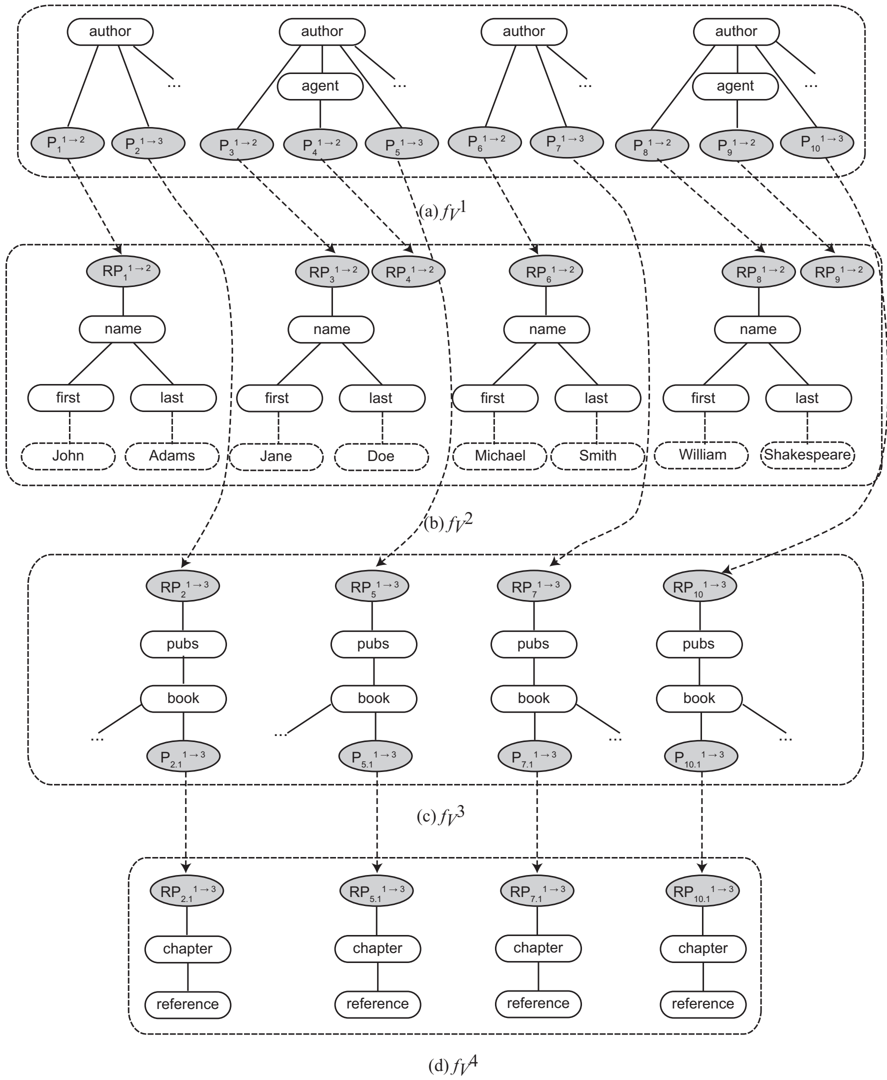

Fig. 17.22 Fragmentation with Proxy Nodes and Numbering

expensive local plan with one of its ancestor fragments until the maximum local plan cost can no longer be reduced.

17.4.4 Optimizing Distributed XML Processing

Research into processing and optimization strategies for distributed execution of XML queries are in their infancy. Although there is active research on a number of fronts and some general methods and principles are emerging, we are far from a full understanding of the issues. In this section we will summarize two areas of research: different distributed execution models focusing on data shipping versus query shipping, and localization and pruning in the case of query shipping systems.

17.4.4.1 Data Shipping versus Query Shipping

Data shipping and query shipping approaches were discussed in Chapter 8 within the context of relational systems. The same choice for distributed query execution exists in the case of XML data management.

One way to execute XML queries over distributed data is to analyze each query to determine the data that it needs, ship those data from their sources to where the query is issued (or to a particular site) and execute the query at that site. This is what is referred to as *data shipping*. XQuery has built-in functionality for data shipping through the fn:doc(URI) function that retrieves the document identified by the URI to the query site and executes the query over the retrieved data. While data shipping is simple to implement and may be useful in certain situations, it only provides inter-query parallelism and cannot exploit intra-query parallel execution opportunities. Furthermore, it relies on the expectation that there is sufficient storage space at each query site to hold the data that are received. Finally, it may require large amounts of data to be moved, posing serious overhead.

The alternative is to execute the query where the data reside. This is called *query shipping* (or *function shipping*). As discussed in Chapter 8, the general approach to query shipping is to decompose the XML query into a set of subqueries and to execute each of these subqueries at the sites where the data reside. Coupled with localization and pruning that we discuss in the next section, this approach provides intra-query parallelism and executes queries where data are located.

Although query shipping is preferable due to its better parallelization properties, it is not easy in the context of XML systems. The fundamental difficulty comes from the fact that, in the most general case, this approach requires shipping both the function and its parameters to a remote site. It is possible that some of the parameters may refer to data at the originating site, requiring the "packaging" of these parameter values and shipping them to a remote site (i.e., call-by-value semantics). If the parameter and return values are atomic, then this is not a problem, but they may be more complex, involving element nodes. This issue also arises in the context of distributed object database systems and we alluded to them in Chapter 15. In the case of XML systems, the serialization of the subtree rooted at the parameter node is packaged and shipped. This raises a number of challenges in XML systems [Zhang et al., 2009b]:

1. In XPath expressions, there may be some axes that are not downward from the parameter node. For example, parent, preceding-sibling (as well as other) axes require accessing data that may not be available in the subtree of the parameter node. A similar problem occurs when certain built-in XQuery functions are executed. For example, root(), id(), idref() functions return nodes that are not descendents of the parameter node, and therefore cannot be executed on the serialization of the subtree rooted at the parameter node.
2. In XML, as in object databases, there is the notion of "identity"; in case of XML, node identity. If two identical nodes are passed as parameters or returned as results, the call-by-value represents them as two different copies, leading to difficulties in node identity comparisons.
3. As noted earlier, in XML there is the notion of document order of nodes and queries are expected to obey this order both in their execution and in their results. The serialization of parameter subtrees in call-by-value organizes nodes with respect to each parameter. Although it is easy to maintain the document order within the serialization of the subtree of each parameter, the relative order of nodes that occur in serializations of different parameters may be different than their order in the original document.
4. There are difficulties with the interaction between different subqueries that access the same document on a given peer. The results of these subqueries would contain nodes from the same document, but ordered differently in the global result.

These problems are still being worked on and general solutions do not yet exist. We describe three quite different approaches to query shipping as indicative of some of the current work.

A proposal to achieve query shipping is to use the theory of partial function evaluation [Buneman et al., 2006; Cong et al., 2007]. Given a function $f(x,y)$, partial evaluation would compute f on one of the inputs (say x) and generate a partial answer, which would be another function f' that is only dependent on the second input y. The way partial evaluation is used to address the issue is to consider the query as a function and the data fragments as its inputs. Then the query can be decomposed into multiple sub-queries, each operating on one fragment. The results of these sub-queries (i.e., functions) are then combined by taking into account the structural relationships between fragments. The overall process, considering an XPath query Q, proceeds as follows:

1. The coordinating site where Q is submitted determines the sites that hold a fragment of the database. Each fragment site and the coordinating site evaluate the query in parallel. At the end of this stage, for some data nodes, the value of each query qualifier is known, while for other nodes, the value of some qualifiers is a Boolean formula whose value is not yet fully determined.
2. In the second phase, the selection part of Q is (partially) evaluated. At the end of this stage, two things are determined for each node n of each fragment: (i)

whether n is part of Q's answer, or (ii) whether or not n is a candidate to be part of Q's answer.

3. In the final phase, the candidate nodes are checked again to determine which ones are indeed part of the answer to Q and any node that is in Q's answer is sent to the coordinating node.

This approach does not decompose the query in the sense that we defined the term. It executes the query over remote fragments, making three passes over each of the fragments. Since it considers only XPath queries, it does not confront the issues related to XQuery that we discussed above.

An alternative that explicitly decomposes the query has been proposed within the context of XRPC project [Zhang and Boncz, 2007; Zhang et al., 2009b]. XRPC extends XQuery by adding remote procedure call functionality through a newly introduced statement execute at {Expr} {FunApp(ParamList)} where Expr is the (explicit or computed) URI of the peer where FunApp() is to be applied.

The target of XRPC is large-scale heterogeneous P2P systems, thus interoperability and efficiency are main design issues. To enable communication between heterogeneous XQuery systems, XRPC also defines an open network protocol called SOAP XRPC that specifies how XDM data types [XDM, 2007] are serialized in XRPC request/response messages. SOAP XRPC protocol encompasses several features to improve efficiency (primarily reducing network latency), by minimizing the number of messages exchanged and the size of message. An important feature of SOAP XRPC is *Bulk RPC* that allows handling of multiple calls to the same function (with different parameters) in a single network interaction. RPC (remote procedure call) is a distributed system functionality that facilitates function calls across different sites. Bulk RPC is exploited when a query contains a function call nested in an XQuery `for`-loop, which, in a naive implementation, would lead to as many RPC network interactions as loop iterations.

The problems with the call-by-value semantics that were discussed above are addressed by a more advanced (but still call-by-copy-based) function parameter passing semantics that is referred to as *call-by-projection* [Zhang et al., 2009b]. Call-by-projection adopts a runtime projection technique to minimize message sizes, which in turn reduces network latency. Basically, it works as follows. A node parameter is first analyzed to see how it is used by the remote function, i.e., a set of *used paths* and a set of *returned paths* of the node parameter are computed. Then, only those descendants of the node parameter, which are actually used by the remote function, are serialized into the request message. At the same time, nodes outside the subtree of the node parameter are added to the request message, if they are needed by the remote function. For instance, if the remote function applies a parent step on the node parameter, the parent node is serialized as well. The same analysis is applied on the function result, so that the remote peer can remove/add nodes into/from the response messages as needed. Thus, the call-by-projection semantics not only preserves node identities and structural properties of XML node parameters (which enables XQuery expressions that access nodes outside the subtrees of remote nodes), but also minimizes message sizes.

Example 17.20. Figure 17.23 shows the impact of the call-by-projection semantics on message sizes and contents.

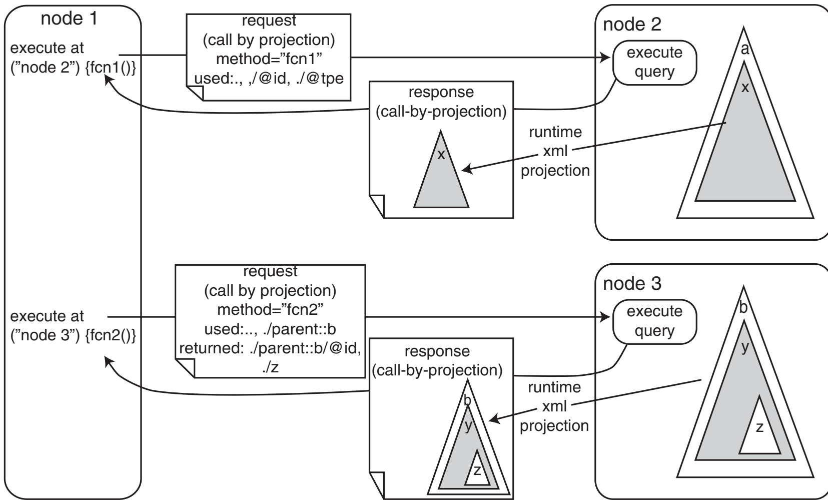

Fig. 17.23 The call-by-projection parameter passing semantics in XPRC

In the upper part of Figure 17.23, node 1 performs an XRPC call to fcn1() on node 2, whose results is the node ⟨x⟩ with a large subtree. With call-by-projection, the query is first analyzed (assuming the call to fcn1() is part of a more complex query) to see how the result of fcn1() is used further in the query. Suppose that only the `id` and `tpe` attributes of ⟨x⟩ are used. This information is included in the request message (shown as "`used:., ./@id, ./@tpe`" in the first request message in the figure). On node 2, before serializing the response message, used paths are applied on the result of fcn1() to compute the projection of ⟨x⟩, which only contains ⟨`x id="..." tpe="..."/`⟩. Finally, the projected node ⟨x⟩ is serialized, resulting in a much smaller response message (compared to serializing the whole node ⟨x⟩).

In the lower part of Figure 17.23, node 1 performs an XRPC call to fcn2() on node 3, whose result is the node ⟨y⟩ with a large subtree. From the second request message, it can be seen that the query containing this call accesses the `parent::b` node of ⟨y⟩ (shown as "`used:., ./parent::b`"), and returns the attributed node `parent::b/@id` and the ⟨z⟩ child nodes of ⟨y⟩ (shown as "`returned:./parent::b/@id, ./z`"). Such a call would not be correctly handled using call-by-value, due to the `parent` step, ♦

The final query shipping approach that we describe focuses on decomposing queries over horizontally and vertically fragmented XML databases as described above [Kling et al., 2010]. This work only addresses XPath queries, and therefore does not deal with the complications introduced by full XQuery decomposition that we discussed above. We describe it only for the case of vertical fragmentation since

that is more interesting (handling horizontal fragmentation is easier). It starts with the QTP representation of the global query (let us call this GQTP) and directly follows the schema graph to get a set of subqueries (i.e., local QTPs – LQTPs), each of which consists of pattern nodes that match items in the same fragment. A child edge from a GQTP node a that corresponds to a document node in fragment f_i to a node b that corresponds to a document node in fragment f_j is replaced by (i) and edge $a \rightarrow P_k^{i \rightarrow j}$, and (ii) an edge $RP_k^{i \rightarrow j} \rightarrow b$. The proxy/root proxy nodes have the same ID, so they establish the connection between a and b. These nodes are marked as extraction points because they are needed to join the results of local QTPs to generate the final result. As with the document fragments, the QTPs form a tree connected by proxy/root proxy nodes. Thus, the usual notions of root/parent/child QTP can be easily defined

Example 17.21. Consider the following XPath query to find references to the books published by "William Shakespeare":

```
/author[name[.//first = 'William' and
    last = 'Shakespeare']]//book//reference
```

This query can be represented by the global QTP of Figure 17.24.

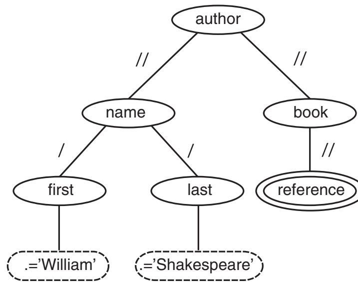

Fig. 17.24 Example QTP

The decomposition of this query based on the vertical fragmentation given in Example 17.18 should result in `author` node being in one subquery (QTP-1), the subtree rooted at `name` being in a second subquery (QTP-2), `book` being in a third subquery (QTP-3), and `reference` in the fourth subquery (QTP-4) as shown in Figure 17.25. ♦

In this approach, each of the QTPs potentially corresponds to a local query plan that can be executed at one site. The issues that we discuss in the next section address concerns related to the optimization of distributed execution of these local plans.

In addition to pure data shipping and pure query shipping approaches discussed above, it is possible to have a hybrid execution model. Active XML that we discussed earlier is an example. It packages each function with the data that it operates on and when the function is encountered in an Active XML document, it is executed remotely where the data reside. However, the result of the function execution is returned to the original active XML site (i.e., data shipping) for further processing.

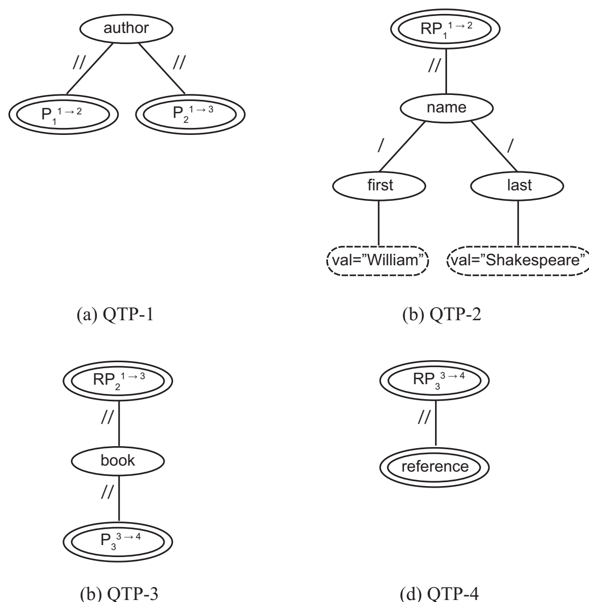

Fig. 17.25 Subqueries after Decomposition

17.4.4.2 Localization and Pruning

As we discussed in Chapter 3, the main objective of localization and pruning is to eliminate unnecessary work by ensuring that the decomposed queries are executed only over the fragments that have data to contribute to the result. Recall that localization was performed by replacing each reference to a global relation with a *localization program* that shows how the global relation can be reconstructed from its fragments. This produces the initial (naïve) query plan. Then, algebraic equivalence rules are used to re-arrange the query plan in order to perform as many operators as possible over each fragment. The localization program is different, of course, for different types of fragmentation. We will follow the same approach here, except that there are further complications in XML databases that are due to the complexity of the XML data model and the XQuery language. As indicated earlier, the general case of distributed execution of XQuery with full power of XML data model is not yet fully solved. To demonstrate localization and pruning more concretely, we will consider a restricted query model and a particular approach proposed by Kling et al. [2010].

In this particular approach, a number of assumptions are made. First, the query plans are represented as QTPs rather than operator trees. Second, queries can have multiple extraction points (i.e., query results are comprised of tuples that consist of multiple nodes), which come from thee same document. Finally, as in XPath, the structural constraints in the queries do not refer to nodes in multiple documents.

Although this is a restricted query model, it is general enough to represent a large class of XPath queries.

Let us first consider a horizontally fragmented XML database. Based on the horizontal fragmentation definition given above, and the query model as specified, the localization program would be the union of fragments – the same as in the relational case. More precisely, given a horizontal fragmentation F_H of database D (i.e., $F_H = f_1, \ldots, f_n$),

$$D = \bigcup_{f_i \in F_H} f_i$$

More interestingly, however, is the definition of the result of a query over a fragmented database, i.e., an initial (or naïve distributed plan). If q is a plan that evaluates the query on an unfragmented database D and F_H is as defined above, then a naïve plan $q(F_H)$ can be defined as

$$q(F_H) := \mathrm{sort}(\bigodot_{f_i \in F_H} q(f_i))$$

where $\odot$ denotes concatenation of results and q_i is the subquery that executes on fragment f_i. It may be necessary to sort the results received from the individual fragments in order to return them in a stable global order as required by the query model.

This naïve plan will access every fragment, which is what pruning attempts to avoid. In this case, since the queries and fragmentation predicates are both represented in the same format (QTP and multiple FTPs, respectively), pruning can be performed by simultaneously traversing these trees and checking for contradictory constraints. If a contradiction is found between the QTP and a FTP_i, there cannot be any result for the query in the fragment corresponding to FTP_i, and the fragment can be eliminated from the distributed plan. This can be achieved by using one of a number of XML tree pattern evaluation algorithms, which we will not get into in this chapter.

Example 17.22. Consider the query given in Example 17.21 and its QTP representation depicted in Figure 17.24.

Assuming the horizontal fragmentation given in Example 17.17, it is clear that this query only needs to run on the fragment that has authors whose last names start with "S" and all other fragments can be eliminated. ♦

In the case of vertical fragmentation, the localization program is (roughly) the equijoin of the subqueries on fragments where the join predicate is defined on the IDs of the proxy/remote proxy pair. More precisely, given $P = \{p_1, \ldots, p_n\}$ as a set of local query plans corresponding to a query q, and F_V as a vertical fragmentation of a document D (i.e., $F_V = \{f_1, \ldots, f_n\}$) such that f_i denotes the vertical fragments corresponding to p_i, the naïve plan can be defined recursively as follows. If $P' \subseteq P$, then $G_{P'}$ is a vertical execution plan for P' if and only if

1. $P' = \{p_i\}$ and $G'_P = p_i$, or

2. $P' = P'_a \cup P'_b, P_a \cap P_b = \emptyset; p_i \in P_a, p_j \in P_b, p_i = parent(p_j); G_{P'_a}$ and $G_{P'_b}$ are vertical execution plans for P'_a and P'_b, respectively; and $G_{P'} = G_{P'_a} \bowtie_{P_*^{i\to j} = RP_*^{i\to j}} G_{P'_b}$.

If G_P is a vertical execution plan for P (the entire set of local query plans), then $G_q = G_P$ is a vertical execution plan for p.

A vertical execution plan must contain all the local plans corresponding to the query. As shown in the recursive definition above, an execution plan for a single local plan is simply the local plan itself (condition 1). For a set of multiple local plans P', it is assumed that P'_a and P'_b are two non-overlapping subsets of P' such that $P'_a \cup P'_b = P'$. Of course, it is necessary that P'_a contains the parent local plan p_i for some local plan p_j in P'_b. An execution plan for P' is then defined by combining execution plans for P'_a and P'_b using a join whose predicate compares the IDs of proxy nodes in the two fragments (condition 2). This is referred to as the *cross-fragment join* [Kling et al., 2010].

Example 17.23. Let p_a, p_b, p_c and p_d represent local plans that evaluate the QTPs shown in Figures 17.25(a), (b), (c) and (d), respectively. The initial vertical plan is given in Figure 17.26 where QTP_i:P_j refers to the proxy node P_j in QTP_i. ♦

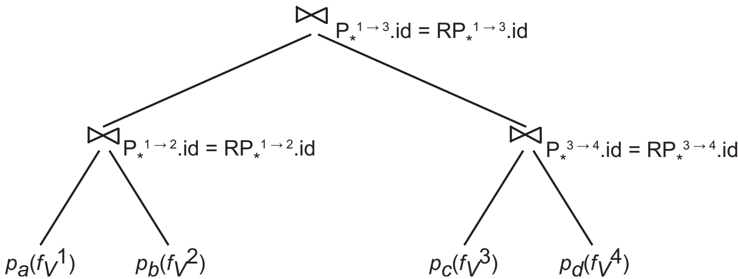

Fig. 17.26 Initial Vertical Plan

If the global QTP does not reach a certain fragment, then the localized plan derived from the local QTPs will not access this fragment. Therefore, the localization technique eliminates some vertical fragments even without further pruning. The partial function execution approach that we introduced earlier works similarly and avoids accessing fragments that are not necessary. However, as demonstrated by Example 17.23, intermediate fragments have to be accessed even if no constraints are evaluated on them. In our example, we have to evaluate QTP_3, and, therefore access fragment f_V^3 (although there is no predicate in the query that refers to any node in that fragment) in order to determine, for example, the root proxy node $RP_3^{1\to4}$ instance in fragment f_V^4 that is a descendent of a particular proxy node $P_*^{1\to4}$ instance in f_V^1.

A way to prune unnecessary fragments from consideration is to store information in the proxy/root proxy nodes that allow identification of all ancestor proxy nodes

for any given root proxy node [Kling et al., 2010]. A simple way of storing this information is by using a Dewey numbering scheme to generate the IDs for each proxy pair. Then it is possible to determine, for any a root proxy node in f_V^4, which proxy node in f_V^1 is its ancestor. This, in turn, would allow answering the query without accessing f_V^3 or evaluating local QTP_3. The benefits of this are twofold: it reduces load on the intermediate fragments (since they are not accessed) and it avoids the cost of computing intermediate results and joining them together.

The numbering scheme works as follows:

1. If a document snippet is in the root fragment, then the proxy nodes in this fragment, and the corresponding root proxy nodes in other fragments are assigned simple numeric IDs.
2. If a document snippet is rooted at a root proxy node, then the ID of each of its proxy nodes is prefixed by the ID of the root proxy node of this document snippet, followed by a numeric identifier that is unique within this snippet.

Example 17.24. Consider the vertical fragmentation given in Figure 17.21. With the introduction of proxy/root proxy pairs and the appropriate numbering as given above, the resulting fragmentation is given in Figure 17.22. The proxy nodes in root fragment f_V^1 are simply numbered. Fragments f_V^2, f_V^3 and f_V^4 consist of document snippets that are rooted at a root proxy. However, of these, only fragment f_V^3 contains proxy nodes, requiring appropriate numbering. ♦

If all proxy/remote proxy pairs are numbered according to this scheme, a root proxy node in a fragment is the descendant of a proxy node at another fragment precisely when the ID of the proxy node is a prefix of the ID of the root proxy node. When evaluating query patterns, this information can be exploited by removing local QTPs from the distributed query plan if they contain no value or structural constraints and no extraction point nodes other than those corresponding to proxies. These local QTPs are only needed to determine whether a root proxy node in some other fragment is a descendant of a proxy node in a third fragment, which can now be inferred from the IDs.

Example 17.25. The initial query plan in Figure 17.26 is now pruned to the plan in Figure 17.27. ♦

17.5 Conclusion

The web has become a major repository of data and documents, making it an important topic to study. As noted earlier, there is no unifying framework for many of the topics that fall under web data management. In this chapter, we focused on three major topics, namely, web search, web querying, and distributed XML data management. Even in these areas, many open problems remain.

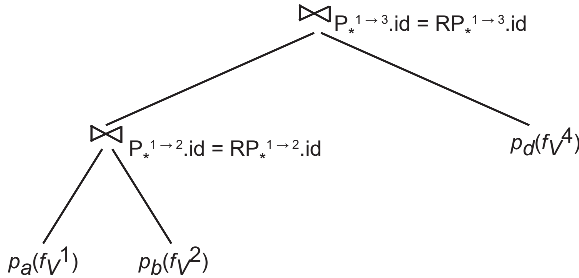

Fig. 17.27 Skipping Vertical Fragments

There are a number of other issues that could be covered. These include service-oriented computing, web data integration, web standards, and others. While some of these have settled, others are still active areas of research. Since it is not possible to cover all of these in detail, we have chosen to focus on the issues related to data management.

17.6 Bibliographic Notes

There are a number of good sources on web topics, each focusing on a different topic. A web data warehousing perspective is given in [Bhowmick et al., 2004]. [Bonato, 2008] primarily focuses on the modelling 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]. There are many books on XML, but a good starting point is [Katz et al., 2004].

A very good overview of web search issues is [Arasu et al., 2001], which we also follow in Section 17.2. In construction of Sections 17.4.1 and 17.4.2, we adopted material from Chapter 2 of [Zhang, 2006]. The discussion of distributed XML follows [Kling et al., 2010] and uses material from Chapter 2 of [Zhang, 2010].

Exercises

Problem 17.1 (**). Consider the graph in Figure 17.28. A node P_i is said to be a *reference* for for node P_j iff there exists an edge from P_j to P_i ($P_j \rightarrow P_i$) and there exist a node P_k such that $P_i \rightarrow P_k$ and $P_j \rightarrow P_k$.

(a) Indicate the reference nodes for each node in the graph.

(b) Find the cost of compressing each node using the formula given in [Adler and Mitzenmacher, 2001] for each of its reference nodes.

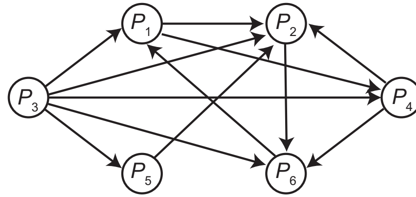

Fig. 17.28 Figure for Problem 17.1

(c) Assuming that (i) for each node we only choose one reference node, and (ii) there must not be cyclic references in the final result, find the optimal set of references that maximizes compression. (Hint: note that this can be systematically done by creating a root node r, and letting all the nodes in the graph point to r, and then finding the minimum spanning tree starting from $r(cost(P_x, r) = \lceil log\ n \rceil * out_deg(P_x))$.)

Problem 17.2. How does web search differ from web querying?

Problem 17.3 (**). Consider the generic search engine architecture in Figure 17.4. 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 17.4 (**). Consider your solution in Problem 17.3. 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 17.5 (*). To increase locality of access and performance in different geographical regions, propose an extension of the web site architecture in Problem 17.4 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 17.6 (*). Consider your solution in Problem 17.5. 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 17.7 (**). Given an XML document modeled as tree, write an algorithm that matches simple XPath expression that only contains child axes and no branch predicates, For example, /A/B/C should return all C elements who are children of some B elements who are in turn the children of the root element A. Note that A may contain child element other than B, and such is true for B as well.

Problem 17.8 (**). 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.

Problem 17.9. Consider the QTP and the set of FTPs shown in Figure 17.29 and the vertical fragmentation schema in Figure 17.20. Determine the fragment(s) that can be excluded from the distributed query plan for this QTP.

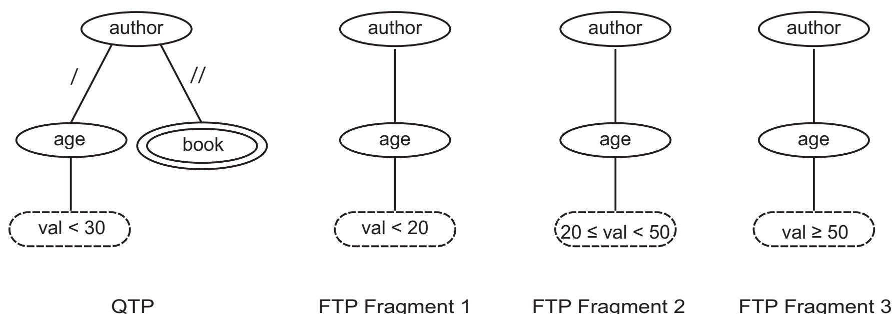

Fig. 17.29 Figure for Problem 17.9

Problem 17.10 (**). Consider the QTP and the FTP shown in Figure 17.30. Can we exclude the fragment defined by this FTP from a query plan for the QTP? Explain your answer

Problem 17.11 (*). Localize the QTP shown in Figure 17.31 for distributed evaluation based on the vertical fragmentation schema shown in Figure 17.20.

Problem 17.12 (**). When evaluating the query from Problem 17.11, can any of the fragments be skipped using the method based on the Dewey decimal system? Explain your answer.

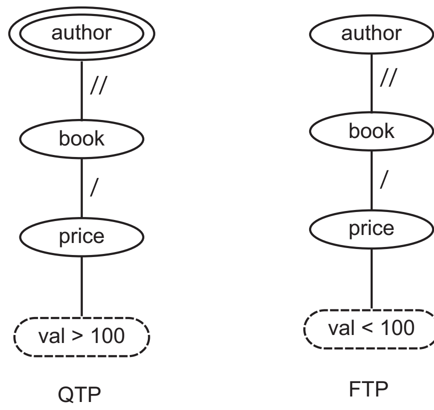

Fig. 17.30 Figure for Problem 17.10

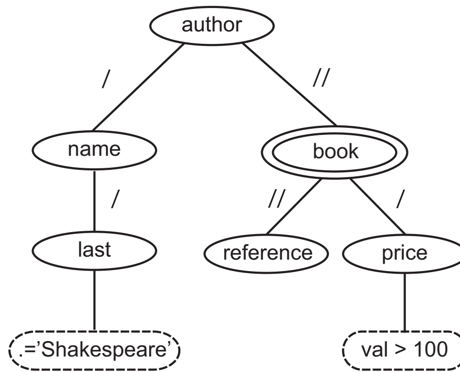

Fig. 17.31 Figure for Problem 17.11