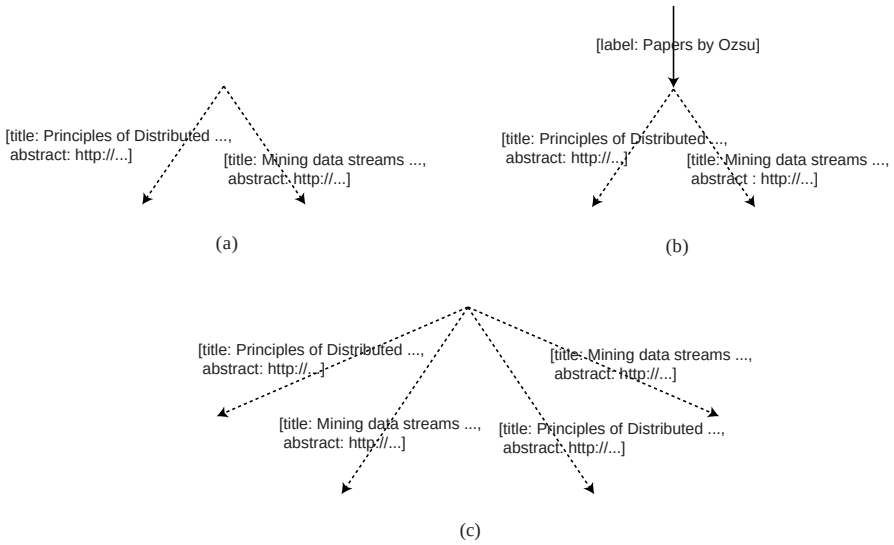


Fig. 17.9 Examples of Hang and Concatenate Operators

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,

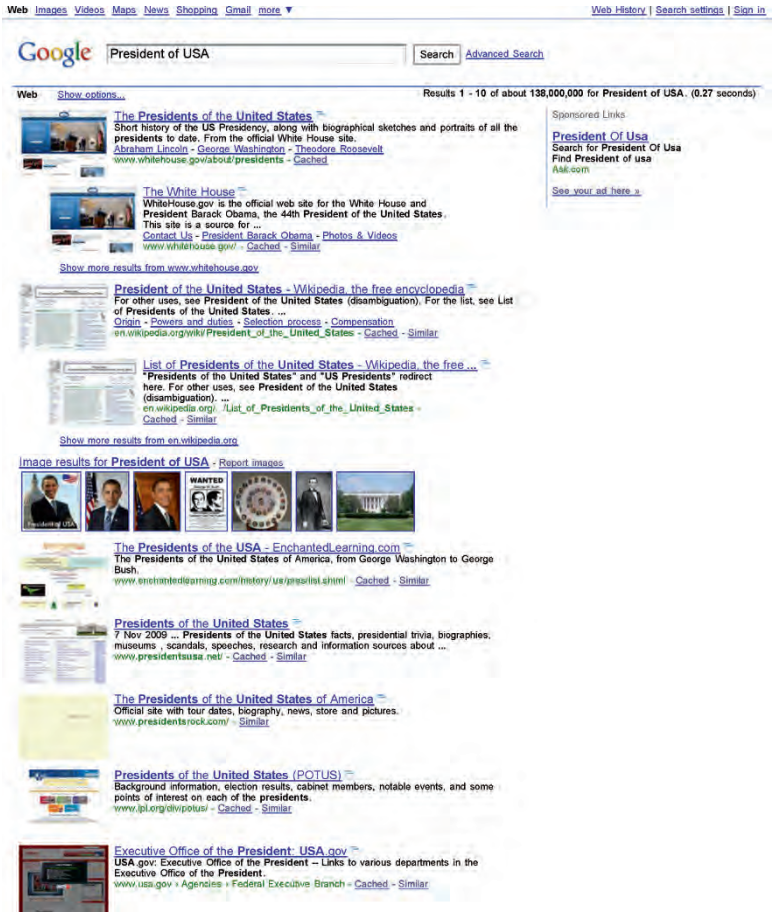


Fig. 17.10 Keyword Search Example

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

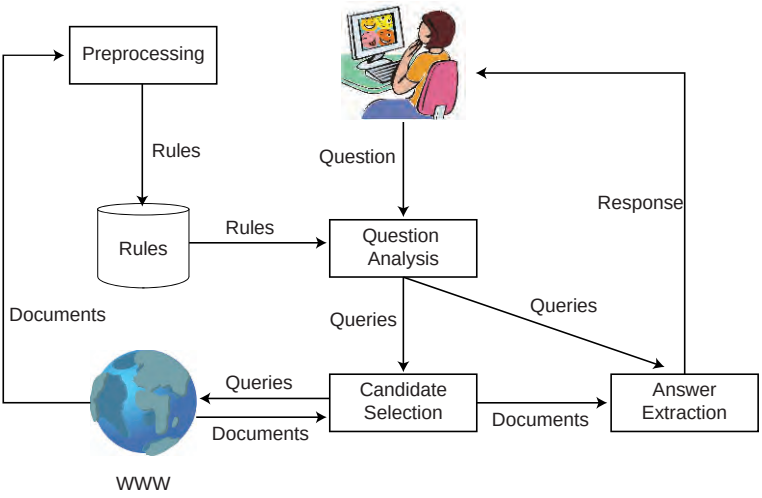


Fig. 17.11 General architecture of QA Systems

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.

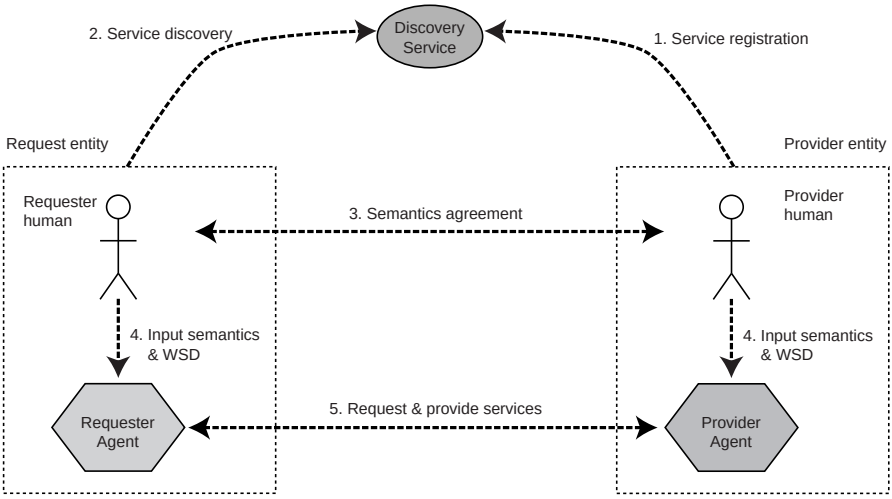


Fig. 17.12 A typical Web Service workflow suggested by the W3C Web Services Architecture (Based on [Booth et al., 2004].)

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

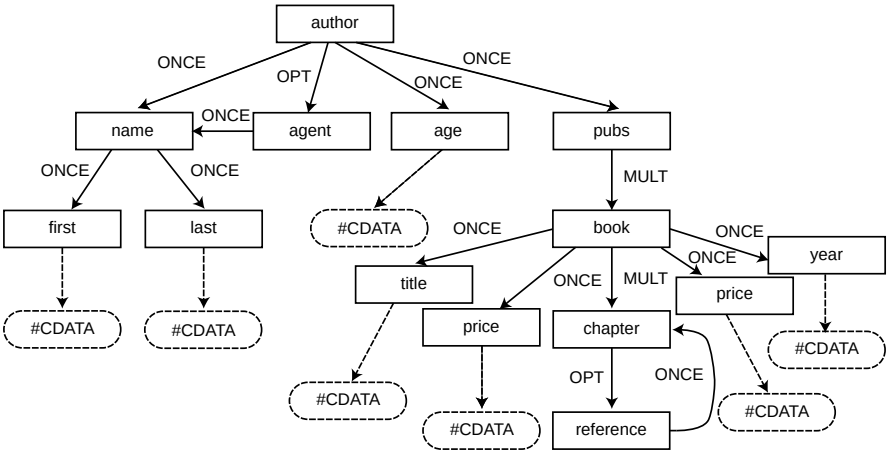


Fig. 17.15 Example XML Schema Graph for Fragmentation

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:

$$
\begin{aligned}
\text{Path} &::= \text{Step}\ (\text{“/”Step})^* \\
\text{Step} &::= \text{axis“ :: ”NameTest}\ (\text{Qualifier})^* \\
\text{NameTest} &::= \text{ElementName} \mid \text{AttributeName} \mid \text{“}*\text{”} \\
\text{Qualifier} &::= \text{‘‘[”Expr“]”} \\
\text{Expr} &::= \text{Path}\ (\text{Comp Atomic})? \\
\text{Comp} &::= \text{“}=\text{”}\mid\text{“}>\text{”}\mid\text{“}<\text{”}\mid\text{“}>=\text{”}\mid\text{“}<=\text{”}\mid\text{“}!=\text{”} \\
\text{Atomic} &::= \text{“’”String“’”}
\end{aligned}
$$

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

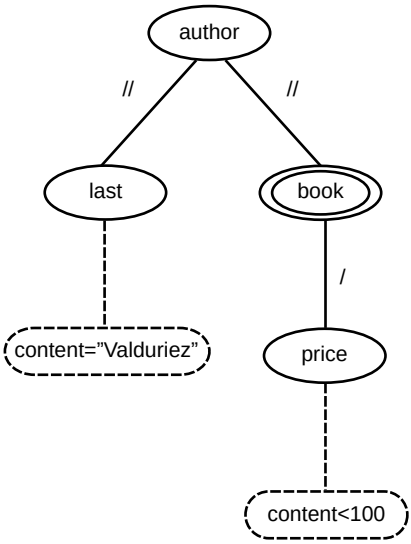


Fig. 17.17 A QTP of expression `/author[.//last = "Valduriez"]//book[price < 100]`

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>
```

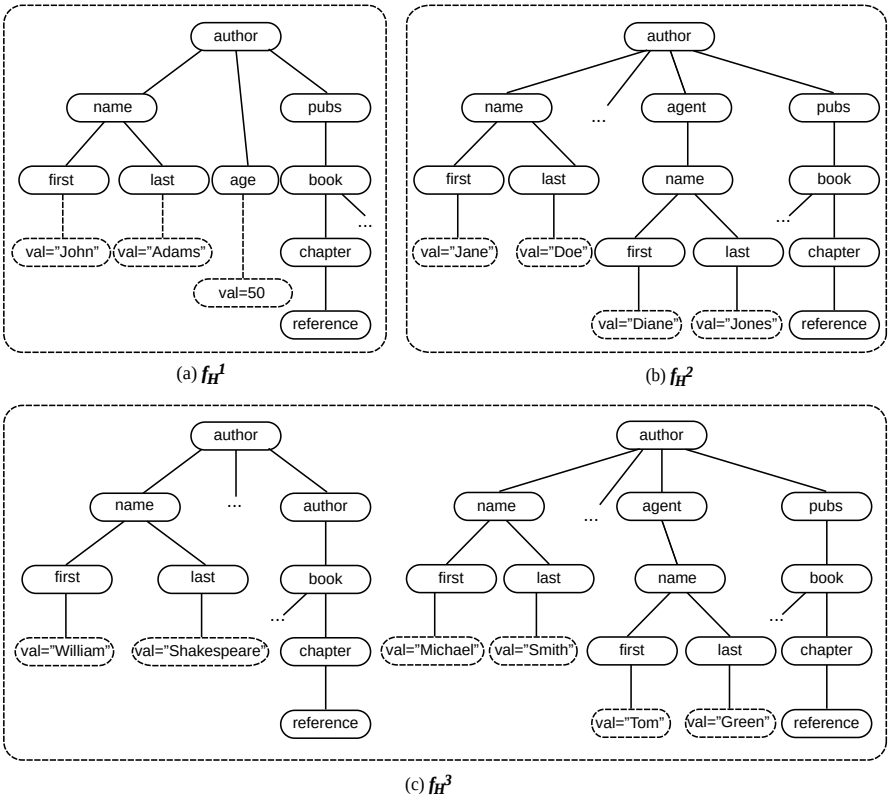
♦

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.



(a) f_H^1 (b) f_H^2

(c) f_H^3

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, $m(\texttt{last})$ 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. ♦

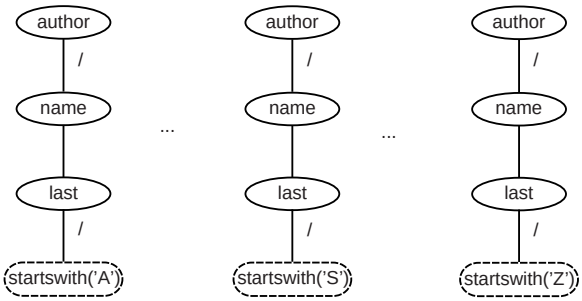


Fig. 17.19 Example Fragmentation Tree Patterns

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 $RP_k^{i \rightarrow j}$

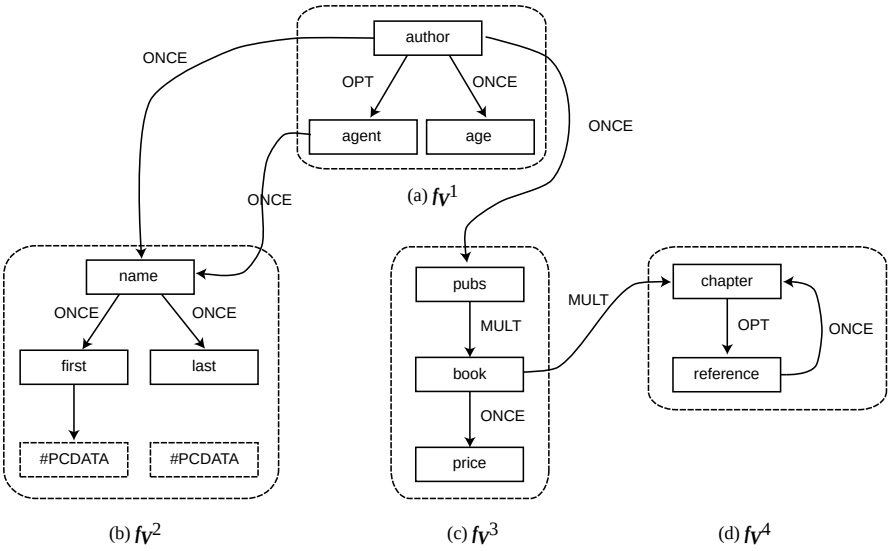


Fig. 17.20 Example Vertical Fragmentation of Schema

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

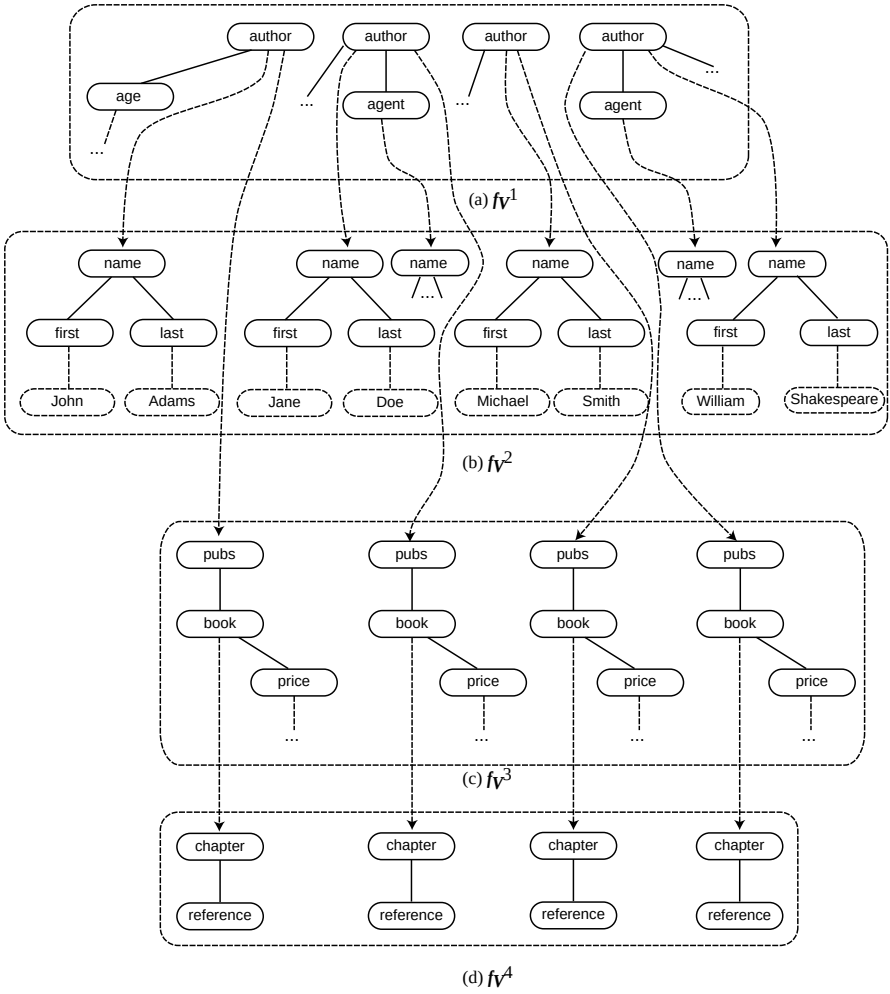


Fig. 17.21 Example Vertical Fragmentation Instances

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 n^{th} 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

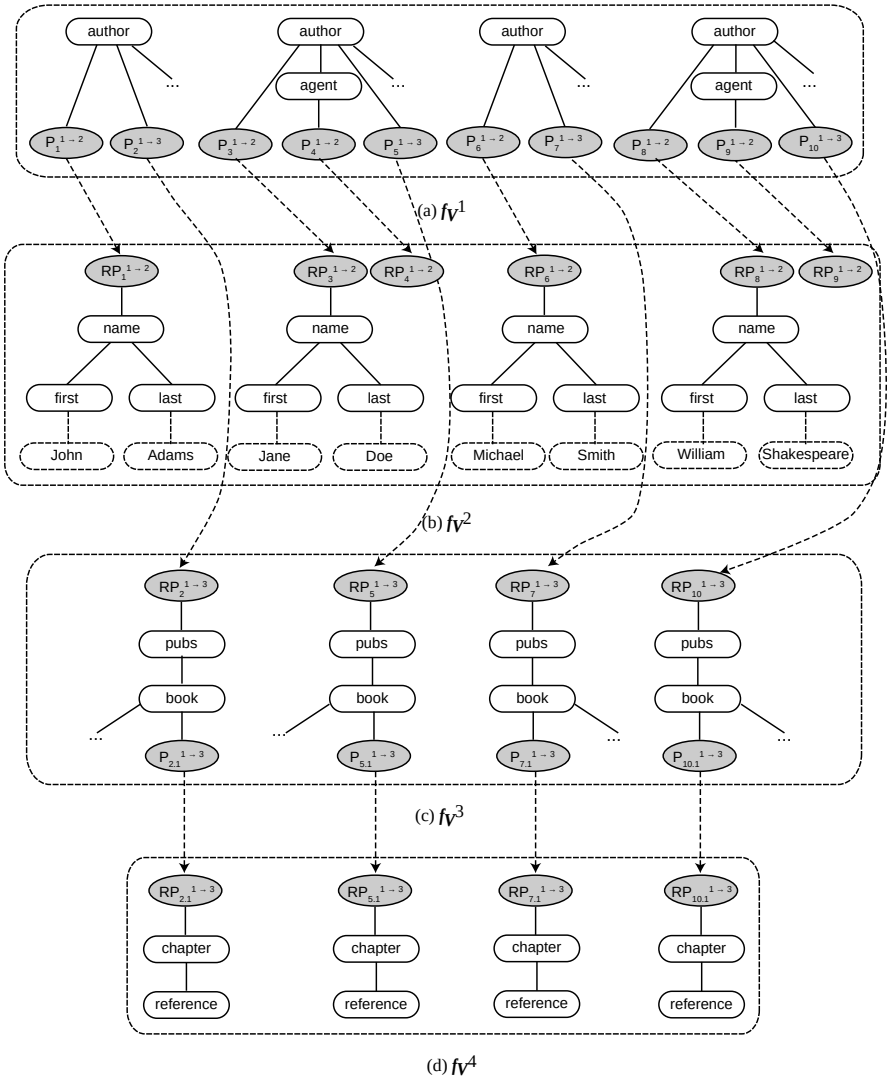


Fig. 17.22 Fragmentation with Proxy Nodes and Numbering

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.

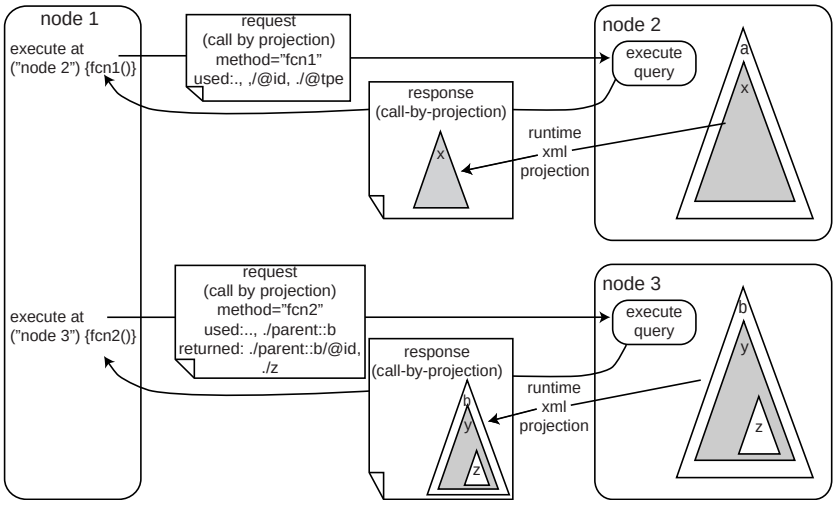


Fig. 17.23 The call-by-projection parameter passing semantics in XPRC

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.

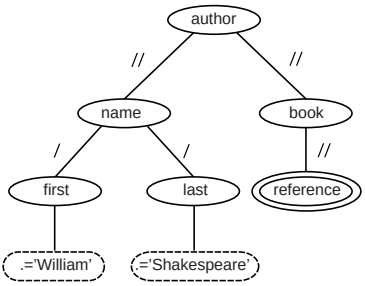
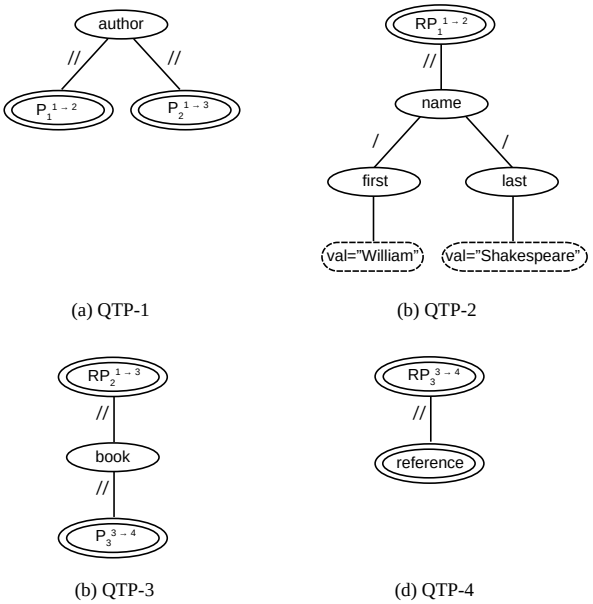


Fig. 17.24 Example QTP

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.



(a) QTP-1 (b) QTP-2

(b) QTP-3 (d) QTP-4

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

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

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. ♦

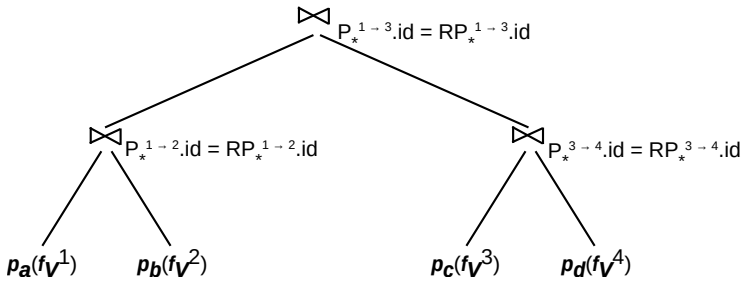


Fig. 17.26 Initial Vertical Plan

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 $RP_3^{1 \to 4}$ instance in fragment f_V^4 that is a descendent of a particular proxy node $P_*^{1 \to 4}$ 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.

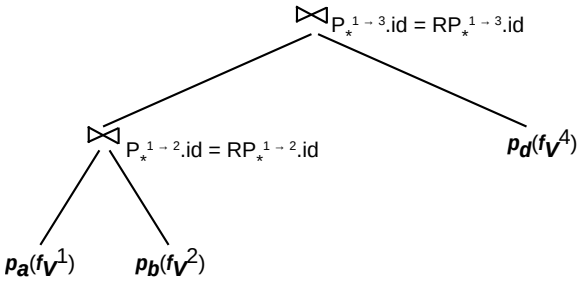


Fig. 17.27 Skipping Vertical Fragments

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.

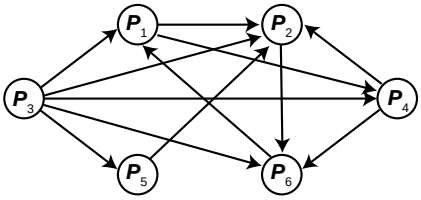


Fig. 17.28 Figure for Problem 17.1

(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.

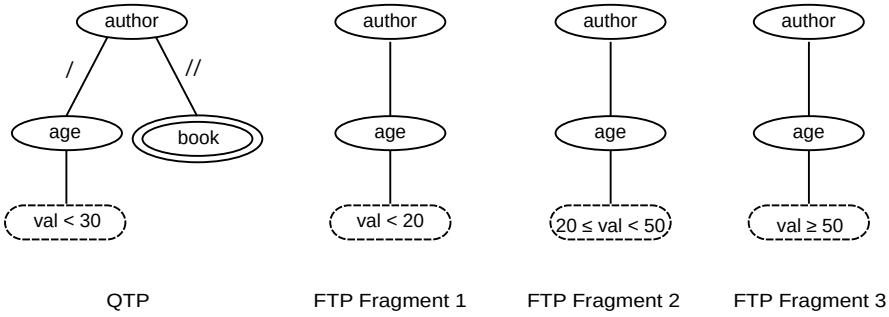


Fig. 17.29 Figure for Problem 17.9

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.

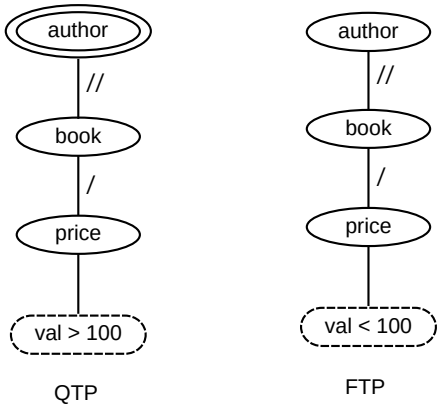


Fig. 17.30 Figure for Problem 17.10

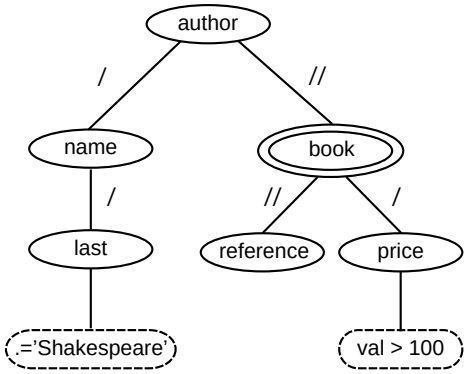


Fig. 17.31 Figure for Problem 17.11

Chapter 18
Current Issues: Streaming Data and Cloud Computing

In this chapter we discuss two topics that are of growing importance in database management. The topics are data stream management (Section 18.1) and cloud data management (Section 18.2). Both of these topics have been topics of considerable interest in the community in recent years. They are still evolving, but there is a possibility that they may have considerable commercial impact. Our objective in this chapter is to give a snapshot of where the field is with respect to these systems at this point, and discuss potential research directions.

18.1 Data Stream Management

The database systems that we have discussed until now consist of a set of unordered objects that are relatively static, with insertions, updates and deletions occurring less frequently than queries. They are sometimes called *snapshot databases* since they show a snapshot of the values of data objects at a given point in time. Queries over these systems are executed when posed and the answer reflects the current state of the database. In these systems, typically, the data are persistent and queries are transient.

However, the past few years have witnessed an emergence of applications that do not fit this data model and querying paradigm. These applications include, among others, sensor networks, network traffic analysis, financial tickers, on-line auctions, and applications that analyze transaction logs (such as web usage logs and telephone call records). In these applications, data are generated in real time, taking the form of an unbounded sequence (stream) of values. These are referred to as the *data stream* applications. In this section, we discuss systems that support these applications; these systems are referred to as *data stream management systems* (DSMS).

A fundamental assumption of the data stream model is that new data are generated continually and in fixed order, although the arrival rates may vary across applications from millions of items per second (e.g., Internet traffic monitoring) down to several items per hour (e.g., temperature and humidity readings from a weather monitoring station). The ordering of streaming data may be implicit (by arrival time at the

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*,
DOI 10.1007/978-1-4419-8834-8_18,

processing site) or explicit (by generation time, as indicated by a *timestamp* appended to each data item by the source). As a result of these assumptions, DSMSs face the following novel requirements.

1. Much of the computation performed by a DSMS is push-based, or data-driven. Newly arrived stream items are continually (or periodically) pushed into the system for processing. On the other hand, a DBMS employs a mostly pull-based, or query-driven computation model, where processing is initiated when a query is posed.
2. As a consequence of the above, DSMS queries are *persistent* (also referred to as continuous, long-running, or standing queries) in that they are issued once, but remain active in the system for a possibly long period of time. This means that a stream of updated results must be produced as time goes on. In contrast, a DBMS deals with one-time queries (issued once and then "forgotten"), whose results are computed over the current state of the database.
3. The system conditions may not be stable during the *lifetime* of a persistent query. For example, the stream arrival rates may fluctuate and the query workload may change.
4. A data stream is assumed to have unbounded, or at least unknown, length. From the system's point of view, it is infeasible to store an entire stream in a DSMS. From the user's point of view, recently arrived data are likely to be more accurate or useful.
5. New data models, query semantics and query languages are needed for DSMSs in order to reflect the facts that streams are ordered and queries are persistent.

The applications that generate streams of data also have similarities in the type of operations that they perform. We list below a set of fundamental continuous query operations over streaming data.

- **Selection:** All streaming applications require support for complex filtering.
- **Nested aggregation:** Complex aggregates, including nested aggregates (e.g., comparing a minimum with a running average) are needed to compute trends in the data.
- **Multiplexing and demultiplexing:** Physical streams may need to be decomposed into a series of logical streams and conversely, logical streams may need to be fused into one physical stream (similar to group-by and union, respectively).
- **Frequent item queries:** These are also known as *top-k* or *threshold* queries, depending on the cut-off condition.
- **Stream mining:** Operations such as pattern matching, similarity searching, and forecasting are needed for on-line mining of streaming data.
- **Joins:** Support should be included for multi-stream joins and joins of streams with static meta-data.

- **Windowed queries:** All of the above query types may be constrained to return results inside a window (e.g., the last 24 hours or the last one hundred packets).

Proposed data stream systems resemble the abstract architecture shown in Figure 18.1. An input monitor regulates the input rates, perhaps by dropping items if the system is unable to keep up. Data are typically stored in three partitions: temporary working storage (e.g., for window queries that will be discussed shortly), summary storage for stream synopses, and static storage for meta-data (e.g., physical location of each source). Long-running queries are registered in the query repository and placed into groups for shared processing, though one-time queries over the current state of the stream may also be posed. The query processor communicates with the input monitor and may re-optimize the query plans in response to changing input rates. Results are streamed to the users or temporarily buffered. Users may then refine their queries based on the latest results.

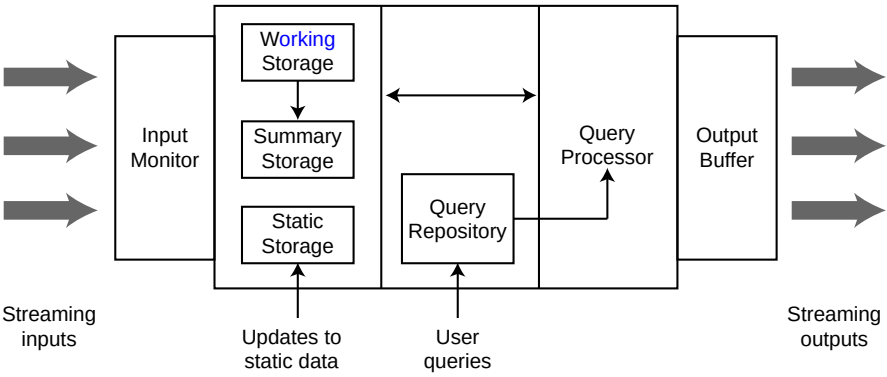


Fig. 18.1 Abstract reference architecture for a data stream management system.

18.1.1 Stream Data Models

A data stream is an append-only sequence of timestamped items that arrive in some order [Guha and McGregor, 2006]. While this is the commonly accepted definition, there are more relaxed versions; for example, *revision tuples*, which are understood to replace previously reported (presumably erroneous) data [Ryvkina et al., 2006], may be considered so that the sequence is not append-only. In publish/subscribe systems, where data are produced by some sources and consumed by those who subscribe to those data feeds, a data stream may be thought of as a sequence of events that are being reported continually [Wu et al., 2006]. Since items may arrive in bursts, a stream may instead be modeled as a sequence of sets (or bags) of elements [Tucker et al., 2003], with each set storing elements that have arrived during the same

unit of time (no order is specified among tuplesthat have arrived at the same time). In relation-based stream models (e.g., STREAM [Arasu et al., 2006]), individual items take the form of relational tuples such that all tuples arriving on the same stream have the same schema. In object-based models (e.g., COUGAR [Bonnet et al., 2001] and Tribeca [Sullivan and Heybey, 1998]), sources and item types may be instantiations of (hierarchical) data types with associated methods. Stream items may contain explicit source-assigned timestamps or implicit timestamps assigned by the DSMS upon arrival. In either case, the timestamp attribute may or may not be part of the stream schema, and therefore may or may not be visible to users. Stream items may arrive out of order (if explicit timestamps are used) and/or in pre-processed form. For instance, rather than propagating the header of each IP packet, one value (or several partially pre-aggregated values) may be produced to summarize the length of a connection between two IP addresses and the number of bytes transmitted. This gives rise to the following list of possible models [Gilbert et al., 2001]:

1. *Unordered cash register*: Individual items from various domains arrive in no particular order and without any pre-processing. This is the most general model.
2. *Ordered cash register*: Individual items from various domains are not pre-processed but arrive in some known order, e.g., timestamp order.
3. *Unordered aggregate*: Individual items from the same domain are pre-processed and only one item per domain arrives in no particular order, e.g., one packet per TCP connection.
4. *Ordered aggregate*: Individual items from the same domain are pre-processed and one item per domain arrives in some known order, e.g., one packet per TCP connection in increasing order of the connection end-times.

As discussed earlier, unbounded streams cannot be stored locally in a DSMS, and only a recent excerpt of a stream is usually of interest at any given time. In general, this may be accomplished using a *time-decay model* [Cohen and Kaplan, 2004; Cohen and Strauss, 2003; Douglis et al., 2004], also referred to as an *amnesic* [Palpanas et al., 2004] or *fading* [Aggarwal et al., 2004] model. Time-decay models discount each item in the stream by a scaling factor that is non-decreasing with time. Exponential and polynomial decay are two examples, as are window models where items within the window are given full consideration and items outside the window are ignored. Windows may be classified according the the following criteria.

1. *Direction of movement of the endpoints:* Two fixed endpoints define a *fixed window*, two sliding endpoints (either forward or backward, replacing old items as new items arrive) define a *window!sliding*, and one fixed endpoint and one moving endpoint (forward or backward) define a *window!landmark*. There are a total of nine possibilities as each of the two endpoints could be fixed, moving forward, or moving backward.

2. *Definition of window size:* Logical, or *time-based* windows are defined in terms of a time interval, whereas physical, (also known as *count-based* or *tuple-based*) windows are defined in terms of the number of tuples. Moreover, *partitioned windows* may be defined by splitting a sliding window into groups and defining a separate count-based window on each group [Arasu et al., 2006]. The most general type is a *predicate window*, in which an arbitrary predicate specifies the contents of the window; e.g., all the packets from TCP connections that are currently open [Ghanem et al., 2006]. A predicate window is analogous to a materialized view.
3. *Windows within windows:* In the *elastic window model*, the maximum window size is given, but queries may need to run over any smaller window within the boundaries of the maximum window [Zhu and Shasha, 2003]. In the *n-of-N window model*, the maximum window size is N tuples or time units, but any smaller window of size n and with one endpoint in common with the larger window is also of interest [Lin et al., 2004].
4. *Window update interval:* Eager updating advances the window upon arrival of each new tuple or expiration of an old tuple, but batch processing (lazy updating) induces a *jumping window*. Note that a count-based window may be updated periodically and a time-based window may be updated after some number of new tuples have arrived; these are referred to as *mixed jumping windows* [Ma et al., 2005]. If the update interval is larger than the window size, then the result is a series of non-overlapping *tumbling windows* [Abadi et al., 2003].

As a consequence of the unbounded nature of data streams, DSMS data models may include some notion of change or drift in the underlying distribution that is assumed to generate the attribute values of stream items [Kifer et al., 2004; Dasu et al., 2006; Zhu and Ravishankar, 2004]. We will come back to this issue when we discuss data stream mining in Section 18.1.8. Additionally, it has been observed that in many practical scenarios, the stream arrival rates and distributions of values tend to be bursty or skewed [Kleinberg, 2002; Korn et al., 2006; Leland et al., 1994; Paxson and Floyd, 1995; Zhu and Shasha, 2003].

18.1.2 Stream Query Languages

Earlier we indicated that stream queries are usually persistent. So, one issue to discuss is what the semantics of these queries are, i.e., how do they generate answers. Persistent queries may be monotonic or non-monotonic. A *monotonic query* is one whose results can be updated incrementally. In other words, if $Q(t)$ is the answer to a query at time t, given two executions of the query at t_i and t_j, $Q(t_i) \subseteq Q(t_j)$ for all $t_j > t_i$. For monotonic queries, one can define the following:

$$Q(t) = \bigcup_{t_i=1}^{t} (Q(t_i) - Q(t_{i-1})) \cup Q(0)$$

That is, it is sufficient to re-evaluate the query over newly arrived items and append qualifying tuples to the result [Arasu et al., 2006]. Consequently, the answer of a monotonic persistent query is a continuous, append-only stream of results. Optionally, the output may be updated periodically by appending a batch of new results. It has been proven that a query is monotonic if and only if it is *non-blocking*, which means that it does not need to wait until the end-of-output marker before producing results [Law et al., 2004].

Non-monotonic queries may produce results that cease to be valid as new data are added and existing data changed (or deleted). Consequently, they may need to be re-computed from scratch during every re-evaluation, giving rise to the following semantics:

$$Q(t) = \bigcup_{t_i=0}^{t} Q(t_i)$$

Let us now consider classes of languages that have been proposed for DSMSs. Three querying paradigms can be identified: declarative, object-based, and procedural. *Declarative languages* have SQL-like syntax, but stream-specific semantics, as described above. Similarly, *object-based languages* resemble SQL in syntax, but employ DSMS-specific constructs and semantics, and may include support for streaming abstract data types (ADTs) and associated methods. Finally, *procedural languages* construct queries by defining data flow through various operators.

18.1.2.1 Declarative Languages

The languages in this class include CQL [Arasu et al., 2006; Arasu and Widom, 2004a], GSQL [Cranor et al., 2003], and StreaQuel [Chandrasekaran et al., 2003]. We discuss each of them briefly.

The Continuous Query Language (CQL) is used in the STREAM DSMS and includes three types of operators: relation-to-relation (corresponding to standard relational algebraic operators), stream-to-relation (*sliding windows*), and relation-to-stream. Conceptually, unbounded streams are converted to relations by way of sliding windows, the query is computed over the current state of the sliding windows as if it were a traditional SQL query, and the output is converted back to a stream. There are three relation-to-stream operators—`Istream`, `Dstream`, and `Rstream`—which specify the nature of the output. The `Istream` operator returns a stream of all those tuples which exist in a relation at the current time, but did not exist at the current time minus one. Thus, `Istream` suggests incremental evaluation of monotonic queries. `Dstream` returns a stream of tuples that existed in the given relation in the previous time unit, but not at the current time. Conceptually, `Dstream` is analogous to generating negative tuples for non-monotonic queries. Finally, the `Rstream`

operator streams the contents of the entire output relation at the current time and corresponds to generating the complete answer of a non-monotonic query. The `Rstream` operator may also be used in periodic query evaluation to produce an output stream consisting of a sequence of relations, each corresponding to the answer at a different point in time.

Example 18.1. Computing a join of two time-based windows of size one minute each, can be performed by the following query:

```
SELECT Rstream(*)
FROM   S1 [RANGE 1 min], S2 [RANGE 1 min]
WHERE  S1.a = S2.a
```

The `RANGE` keyword following the name of the input stream specifies a time-based sliding window on that stream, whereas the `ROWS` keyword may be used to define count-based sliding windows. ♦

GSQL is used in Gigascope, a stream database for network monitoring and analysis. The input and output of each operator is a stream for reasons of composability. Each stream is required to have an ordering attribute, such as timestamp or packet sequence number. GSQL includes a subset of the operators found in SQL, namely selection, aggregation with group-by, and join of two streams, whose predicate must include ordering attributes that form a join window. The *stream merge* operator, not found in standard SQL, is included and works as an order-preserving union of ordered streams. This operator is useful in network traffic analysis, where flows from multiple links need to be merged for analysis. Only landmark windows are supported directly, but sliding windows may be simulated via user-defined functions.

StreaQuel is used in the TelegraphCQ system and is noteworthy for its windowing capabilities. Each query, expressed in SQL syntax and constructed from SQL's set of relational operators, is followed by a for-loop construct with a variable `t` that iterates over time. The loop contains a `WindowIs` statement that specifies the type and size of the window. Let `S` be a stream and let `ST` be the start time of a query. To specify a sliding window over `S` with size five that should run for fifty time units, the following for-loop may be appended to the query.

```
for(t=ST; t<ST+50; t++)
  WindowIs(S, t-4, t)
```

Changing to a landmark window can be done by replacing `t-4` with some constant in the `WindowIs` statement. Changing the for-loop increment condition to `t=t+5` would cause the query to re-execute every five time units. The output of a StreaQuel query consists of a time sequence of sets, each set corresponding to the answer set of the query at that time.

18.1.2.2 Object-Based Languages

One approach to object-oriented stream modeling is to classify stream contents according to a type hierarchy. This method is used in the Tribeca network monitoring

system, which implements Internet protocol layers as hierarchical data types [Sullivan and Heybey, 1998]. The query language used in Tribeca has SQL-like syntax, but accepts a single stream as input, and returns one or more output streams. Supported operators are limited to projection, selection, aggregation over the entire input stream or over a sliding window, multiplex and demultiplex (corresponding to union and group-by respectively, except that different sets of operators may be applied on each of the demultiplexed sub-streams), as well as a join of the input stream with a fixed window.

Another object-based possibility is to model the sources as ADTs, as in the COUGAR system for managing sensor data [Bonnet et al., 2001]. Each type of sensor is modeled by an ADT, whose interface consists of the supported signal processing methods. The proposed query language has SQL-like syntax and also includes a `$every()` clause that indicates the query re-execution frequency. However, few details on the language are available in the published literature and therefore it is not included in Figure 18.2.

Example 18.2. A simple query that runs every sixty seconds and returns temperature readings from all sensors on the third floor of a building may be specified as follows:

```
SELECT R.s.getTemperature()
FROM   R
WHERE  R.floor = 3 AND $every(60)
```

♦

18.1.2.3 Procedural Languages

An alternative to declarative query languages is to let the user specify how the data should flow through the system. In the Aurora DSMS [Abadi et al., 2003], users construct query plans via a graphical interface by arranging boxes, corresponding to query operators, and joining them with directed arcs to specify data flow, though the system may later re-arrange, add, or remove operators in the optimization phase. SQuAl is the boxes-and-arrows query language used in Aurora, which accepts streams as inputs and returns streams as output (however, static data sets may be incorporated into query plans via *connection points* [Abadi et al., 2003]). There are a total of seven operators in the SQuAl algebra, four of them order-sensitive. The three order-insensitive operators are projection, union, and `map`, the last applying an arbitrary function to each of the tuples in the stream or a window thereof. The other four operators require an order specification, which includes the ordered field and a slack parameter. The latter defines the maximum disorder in the stream, e.g., a slack of 2 means that each tuple in the stream is either in sorted order, or at most two positions or two time units away from being in sorted order. The four order-sensitive operators are buffered sort (which takes an almost-sorted stream and the slack parameter, and outputs the stream in sorted order), windowed aggregates (in which the user can specify how often to advance the window and re-evaluate the aggregate), binary band join (which joins tuples whose timestamps are at most t units apart), and resample

(which generates missing stream values by interpolation, e.g., given tuples with timestamps 1 and 3, a new tuple with timestamp 2 can be generated with an attribute value that is an average of the other two tuples' values. Other resampling functions are also possible, e.g., the maximum, minimum, or weighted average of the two neighbouring data values.

18.1.2.4 Summary of DSMS Query Languages

A summary of the proposed DSMS query languages is provided in Figure 18.2 with respect to the allowed inputs and outputs (streams and/or relations), novel operators, supported window types (fixed, landmark or sliding), and supported query re-execution frequency (continuous and/or periodic). With the exception of SQuAl, the surface syntax of DSMS query languages is similar to SQL, but their semantics are considerably different. CQL allows the widest range of semantics with its relation-to-stream operators; note that CQL uses the semantics of SQL during its relation-to-relation phase and incorporates streaming semantics in the stream-to-relation and relation-to-stream components. On the other hand, GSQL, SQuAL, and Tribeca only allow streaming output, whereas StreaQuel continually (or periodically) outputs the entire answer set. In terms of expressive power, CQL closely mirrors SQL as CQL's core set of operators is identical to that of SQL. Additionally, StreaQuel can express a wider range of windows than CQL. GSQL, SQuAl, and Tribeca, which operate in the stream-in-stream-out mode, may be thought of as restrictions of SQL as they focus on incremental, non-blocking computation. In particular, GSQL and Tribeca are application-specific (network monitoring) and have been designed for very fast implementation [Cranor et al., 2003]. However, although SQuAl and GSQL are stream-in/stream-out languages, and, as a result, may have lost some expressive power as compared to SQL, they may regain this power via user-defined functions. Moreover, SQuAl is noteworthy for its attention to issues related to real-time processing such as buffering, out-of-order arrivals and timeouts.

Language/ system	Allowed inputs	Allowed outputs	Novel operators	Supported windows	Execution frequency
CQL/ STREAM	streams and relations	streams and relations	relation-to-stream, stream-to-relation	sliding	continuous or periodic
GSQL/ Gigascope	streams	streams	order-preserving union	landmark	periodic
SQuAl/ Aurora	streams and relations	streams	resample, `map`, buffered sort	fixed, landmark, sliding	continuous or periodic
StreaQuel/ TelegraphCQ	streams and relations	sequences of relations	`WindowIs`	fixed, landmark, sliding	continuous or periodic
Tribeca	single stream	streams	multiplex, demultiplex	fixed, landmark, sliding	continuous

Fig. 18.2 Summary of proposed data stream languages

18.1.3 Streaming Operators and their Implementation

While the streaming languages discussed above may resemble standard SQL, their implementation, processing, and optimization present novel challenges. In this section, we highlight the differences between streaming operators and traditional relational operators, including non-blocking behavior, approximations, and sliding windows. Note that simple relational operators such as projection and selection (that do not keep state information) may be used in streaming queries without any modifications.

Some relational operators are blocking. For instance, prior to returning the next tuple, the Nested Loops Join (NLJ) may potentially scan the entire inner relation and compare each tuple therein with the current outer tuple. Some operators have non-blocking counterparts, such as joins [Haas and Hellerstein, 1999a; Urhan and Franklin, 2000; Viglas et al., 2003; Wilschut and Apers, 1991] and simple aggregates [Hellerstein et al., 1997; Wang et al., 2003c]. For example, a pipelined symmetric hash join [Wilschut and Apers, 1991] builds hash tables on-the-fly for each of the participating relations. Hash tables are stored in main memory and when a tuple from one of the relations arrives, it is inserted into its table and the other tables are probed for matches. It is also possible to incrementally output the average of all the items seen so far by maintaining the cumulative sum and item count. When a new item arrives, the item count is incremented, the new item's value is added to the sum, and an updated average is computed by dividing the sum by the count. There remains the issue of memory constraints if an operator requires too much working memory, so a windowing scheme may be needed to bound the memory requirements. Hashing has also been used in developing join execution strategies over DHT-based P2P systems [Palma et al., 2009].

Another way to unblock query operators is to exploit constraints over the input streams. Schema-level constraints include synchronization among timestamps in multiple streams, clustering (duplicates arrive contiguously), and ordering [Babu et al., 2004b]. If two streams have nearly synchronized timestamps, an equi-join on the timestamp can be performed in limited memory: a *scrambling bound B* may be set such that if a tuple with timestamp τ arrives, then no tuple with timestamp greater than $\tau - B$ may arrive later [Motwani et al., 2003].

Constraints at the data level may take the form of control packets inserted into a stream, called *punctuations* [Tucker et al., 2003]. Punctuations are constraints (encoded as data items) that specify conditions for all future items. For instance, a punctuation may arrive asserting that all the items henceforth shall have the A attribute value larger than 10. This punctuation could be used to partially unblock a group-by query on A since all the groups where $A \leq 10$ are guaranteed not to change for the remainder of the stream's lifetime, or until another punctuation arrives and specifies otherwise. Punctuations may also be used to synchronize multiple streams in that a source may send a punctuation asserting that it will not produce any tuples with timestamp smaller than τ [Arasu et al., 2006].

As discussed above, unblocking a query operator may be accomplished by re-implementing it in an incremental form, restricting it to operate over a window (more on this shortly), and exploiting stream constraints. However, there may be cases

where an incremental version of an operator does not exist or is inefficient to evaluate, where even a sliding window is too large to fit in main memory, or where no suitable stream constraints are present. In these cases, compact stream summaries may be stored and approximate queries may be posed over the summaries. This implies a trade-off between accuracy and the amount of memory used to store the summaries. An additional restriction is that the processing time per item should be kept small, especially if the inputs arrive at a fast rate.

Counting methods, used mainly to compute quantiles and frequent item sets, typically store frequency counts of selected item types (perhaps chosen by sampling) along with error bounds on their true frequencies. Hashing may also be used to summarize a stream, especially when searching for frequent items—each item type may be hashed to n buckets by n distinct hash functions and may be considered a potentially frequent flow if all of its hash buckets are large. Sampling is a well known data reduction technique and may be used to compute various queries to within a known error bound. However, some queries (e.g., finding the maximum element in a stream) may not be reliably computed by sampling.

Sketches were initially proposed by Alon et al. [1996] and have since then been used in various approximate algorithms. Let $f(i)$ be the number of occurrences of value i in a stream. A sketch of a data stream is created by taking the inner product of f with a vector of random values chosen from some distribution with a known expectation. Moreover, wavelet transforms (that reduce the underlying signal to a small set of coefficients) have been proposed to approximate aggregates over infinite streams.

We end this section with a discussion of window operators. Sliding window operators process two types of events: arrivals of new tuples and expirations of old tuples; the orthogonal problem of determining when tuples expire will be discussed in the next section. The actions taken upon arrival and expiration vary across operators [Hammad et al., 2003b; Vossough and Getta, 2002]. A new tuple may generate new results (e.g., join) or remove previously generated results (e.g., negation). Furthermore, an expired tuple may cause a removal of one or more tuples from the result (e.g., aggregation) or an addition of new tuples to the result (e.g., duplicate elimination and negation). Moreover, operators that must explicitly react to expired tuples (by producing new results or invalidating existing results) perform state purging eagerly (e.g., duplicate elimination, aggregation, and negation), whereas others may do so eagerly or lazily (e.g., join).

In a sliding window join, newly arrived tuples on one of the inputs probe the state of the other input, as in a join of unbounded streams. Additionally, expired tuples are removed from the state [Golab and Özsu, 2003b; Hammad et al., 2003a, 2005; Kang et al., 2003; Wang et al., 2004]. Expiration can be done periodically (lazily), so long as old tuples can be identified and skipped during processing.

Aggregation over a sliding window updates its result when new tuples arrive and when old tuples expire. In many cases, the entire window needs to be stored in order to account for expired tuples, although selected tuples may sometimes be removed early if their expiration is guaranteed not to influence the result. For example, when computing `MAX`, tuples with value v need not be stored if there is another tuple in the

window with value greater than v and a younger timestamp. Additionally, in order to enable incremental computation, the aggregation operator stores the current answer (for distributive and algebraic aggregates) or frequency counters of the distinct values present in the window (for holistic aggregates). For instance, computing `COUNT` requires storing the current count, incrementing it when a new tuple arrives, and decrementing it when a tuple expires. In this case, in contrast to the join operator, expirations must be dealt with immediately so that an up-to-date aggregate value can be returned right away.

Duplicate elimination over a sliding window may also produce new output when an input tuple expires. This occurs if a tuple with value v was produced on the output stream and later expires from its window, yet there are other tuples with value v still present in the window [Hammad et al., 2003b]. Alternatively, as is the case in the STREAM system, duplicate elimination may produce a single result tuple with a particular value v and retain it on the output stream so long as there is at least one tuple with value v present in the window. In both cases, expirations must be handled eagerly so that the correct result is maintained at all times.

Finally, negation of two sliding windows, $W_1 - W_2$, may produce *negative tuples* (e.g., arrival of a W_2-tuple with value v causes the deletion of a previously reported result with value v), but may also produce new results upon expiration of tuples from W_2 (e.g., if a tuple with value v expires from W_2, then a W_1-tuple with value v may need to be appended to the output stream [Hammad et al., 2003b]). There are methods for implementing duplicate-preserving negation, but those are beyond our scope in this chapter.

18.1.4 Query Processing

Let us now discuss the issues related to processing queries in DSMSs. The overall process is similar to relational systems: declarative queries are translated into execution plans that map logical operators specified in the query into physical implementations. For now, let us assume that the inputs and operator state fit in main memory; we will discuss disk-based processing later.

18.1.4.1 Queuing and Scheduling

DBMS operators are pull-based, whereas DSMS operators consume data pushed into the plan by the sources.

Queues allow sources to push data into the query plan and operators to retrieve data as needed [Abadi et al., 2003; Adamic and Huberman, 2000; Arasu et al., 2006; Madden and Franklin, 2002; Madden et al., 2002a]. A simple scheduling strategy allocates a time slice to each operator, during which the operator extracts tuples from its input queue(s), processes them in timestamp order, and deposits output tuples into the next operator's input queue. The time slice may be fixed or dynamically

calculated based upon the size of an operator's input queue and/or processing speed. A possible improvement could be to schedule one or more tuples to be processed by multiple operators at once. In general, there are several possibly conflicting criteria involved in choosing a scheduling strategy, among them queue sizes in the presence of bursty stream arrival patterns [Babcock et al., 2004], average or maximum latency of output tuples [Carney et al., 2003; Jiang and Chakravarthy, 2004; Ou et al., 2005], and average or maximum delay in reporting the answer relative to the arrival of new data [Sharaf et al., 2005].

18.1.4.2 Determining When Tuples Expire

In addition to dequeuing and processing new tuples, sliding window operators must remove old tuples from their state buffers and possibly update their answers, as discussed in Section 18.1.3. Expiration from an individual time-based window is simple: a tuple expires if its timestamp falls out of the range of the window. That is, when a new tuple with timestamp ts arrives, it receives another timestamp, call it exp, that denotes its expiration time as ts plus the window length. In effect, every tuple in the window may be associated with a lifetime interval of length equal to the window size [Krämer and Seeger, 2005]. Now, if this tuple joins with a tuple from another window, whose insertion and expiration timestamps are ts' and exp', respectively, then the expiration timestamp of the result tuple is set to $\min(exp, exp')$. That is, a composite result tuple expires if at least one of its constituent tuples expires from its windows. This means that various join results may have different lifetime lengths and furthermore, the lifetime of a join result may have a lifetime that is shorter than the window size [Cammert et al., 2006]. Moreover, as discussed above, the negation operator may force some result tuples to expire earlier than their exp timestamps by generating negative tuples. Finally, if a stream is not bounded by a sliding window, then the expiration time of each tuple is infinity [Krämer and Seeger, 2005].

In a count-based window, the number of tuples remains constant over time. Therefore, expiration can be implemented by overwriting the oldest tuple with a newly arrived tuple. However, if an operator stores state corresponding to the output of a count-based window join, then the number of tuples in the state may change, depending upon the join attribute values of new tuples. In this case, expirations must be signaled explicitly using negative tuples.

18.1.4.3 Continuous Query Processing over Sliding Windows

There are two techniques for sliding window query processing and state maintenance: the negative tuple approach and the direct approach. In the negative tuple approach [Arasu et al., 2006; Hammad et al., 2003b, 2004], each window referenced in the query is assigned an operator that explicitly generates a negative tuple for every expiration, in addition to pushing newly arrived tuples into the query plan. Thus, each window must be materialized so that the appropriate negative tuples

are produced. This approach generalizes the purpose of negative tuples, which are now used to signal all expirations explicitly, rather than only being produced by the negation operator if a result tuple expires because it no longer satisfies the negation condition. Negative tuples propagate through the query plan and are processed by operators in a similar way as regular tuples, but they also cause operators to remove corresponding "real" tuples from their state. The negative tuple approach can be implemented efficiently using hash tables as operator state so that expired tuples can be looked up quickly in response to negative tuples. Conceptually, this is similar to a DBMS indexing a table or materialized view on the primary key in order to speed up insertions and deletions. However, the downside is that twice as many tuples must be processed by the query because every tuple eventually expires from its window and generates a corresponding negative tuple. Furthermore, additional operators must be present in the plan to generate negative tuples as the window slides forward.

Direct approach [Hammad et al., 2003b, 2004] handles negation-free queries over time-based windows. These queries have the property that the expiration times of base tuples and intermediate results can be determined via their *exp* timestamps, as explained in Section 18.1.4.2. Hence, operators can access their state directly and find expired tuples without the need for negative tuples. The direct approach does not incur the overhead of negative tuples and does not have to store the base windows referenced in the query. However, it may be slower than the negative tuple approach for queries over multiple windows [Hammad et al., 2003b]. This is because straightforward implementations of state buffers may require a sequential scan during insertions or deletions. For example, if the state buffer is sorted by tuple arrival time, then insertions are simple, but deletions require a sequential scan of the buffer. On the other hand, sorting the buffer by expiration time simplifies deletions, but insertions may require a sequential scan to ensure that the new tuple is ordered correctly, unless the insertion order is the same as the expiration order.

18.1.4.4 Periodic Query Processing Over Sliding Windows

Query Processing over Windows Stored in Memory.

For reasons of efficiency (reduced expiration and query processing costs) and user preference (users may find it easier to deal with periodic output rather than a continuous output stream [Arasu and Widom, 2004b; Chandrasekaran and Franklin, 2003]), sliding windows may be advanced and queries re-evaluated periodically with a specified frequency [Abadi et al., 2003; Chandrasekaran et al., 2003; Golab et al., 2004; Liu et al., 1999]. As illustrated in Figure 18.3, a periodically-sliding window can be modeled as a circular array of *sub-windows*, each spanning an equal time interval for time-based windows (e.g., a ten-minute window that slides every minute) or an equal number of tuples for tuple-based windows (e.g., a 100-tuple window that slides every ten tuples).

Rather than storing the entire window and re-computing an aggregate after every new tuple arrives or an old tuple expires, a synopsis can be stored that pre-aggregates

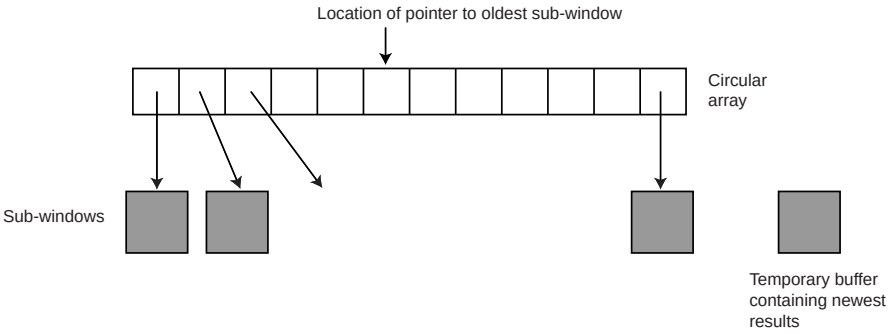


Fig. 18.3 Sliding window implemented as a circular array of pointers to sub-windows

each sub-window and reports updated answers whenever the window slides forward by one sub-window. Thus a "window update" occurs when the oldest sub-window is replaced with newly arrived data (accumulated in a buffer), thereby sliding the window forward by one sub-window. Depending on the type of operator one deals with, it would be necessary to use different types of synopsis (e.g., a *running synopsis* [Arasu and Widom, 2004b] for subtractable aggregates [Cohen, 2006] such as `SUM` and `COUNT` or an *interval synopsis* for distributive aggregates that are not subtractable, such as `MIN` and `MAX`). An aggregate f is subtractable if, for two multi-sets X and Y such that $X \supseteq Y$, $f(X-Y) = f(X) - f(Y)$.Details are beyond our scope in this chapter.

A disadvantage of periodic query evaluation is that results may be stale. One way to stream new results after each new item arrives is to bound the error caused by delayed expiration of tuples in the oldest sub-window. It has been shown [Datar et al., 2002] that restricting the sizes of the sub-windows (in terms of the number of tuples) to powers of two and imposing a limit on the number of sub-windows of each size yields a space-optimal algorithm (called *exponential histogram*, or EH) that approximates simple aggregates to within ε using logarithmic space (with respect to the sliding window size). Variations of the EH algorithm have been used to approximately compute the sum [Datar et al., 2002; Gibbons and Tirthapura, 2002], variance and k-medians clustering [Babcock et al., 2003], windowed histograms [Qiao et al., 2003], and order statistics [Lin et al., 2004; Xu et al., 2004]. Extensions of the EH algorithm to time-based windows have also been proposed [Cohen and Strauss, 2003].

18.1.4.5 Query Processing over Windows Stored on Disk.

In traditional database applications that use secondary storage, performance may be improved if appropriate indices are built. Consider maintaining an index over a periodically-sliding window stored on disk, e.g., in a data warehousing scenario where new data arrive periodically and decision support queries are executed (off-

line) over the latest portion of the data. In order to reduce the index maintenance costs, it is desirable to avoid bringing the entire window into memory during every update. This can be done by partitioning the data so as to localize updates (i.e., insertions of newly arrived data and deletion of tuples that have expired from the window) to a small number of disk pages. For example, if an index over a sliding window is partitioned chronologically [Folkert et al., 2005; Shivakumar and García-Molina, 1997], then only the youngest partition incurs insertions, while only the oldest partition needs to be checked for expirations (the remaining partitions "in the middle" are not accessed).The disadvantage of chronological clustering is that records with the same search key may be scattered across a very large number of disk pages, causing index probes to incur prohibitively many disk I/Os.

One way to reduce index access costs is to store a reduced (summarized) version of the data that fits on fewer disk pages [Chandrasekaran and Franklin, 2004], but this does not necessarily improve index update times. In order to balance the access and update times, a *wave index* has been proposed that chronologically divides a sliding window into n equal partitions, each of which is separately indexed and clustered by search key for efficient data retrieval [Shivakumar and García-Molina, 1997]. The window can be partitioned either by insertion time or by expiration time; these are equivalent from the perspective of wave indexes.

18.1.5 DSMS Query Optimization

It is usually the case that a query may be executed in a number of different ways. A DBMS query optimizer is responsible for enumerating (some or all of) the possible query execution strategies and choosing an efficient one using a cost model and/or a set of transformation rules. A DSMS query optimizer has the same responsibility, but it must use an appropriate cost model and rewrite rules. Additionally, DSMS query optimization involves adaptivity, load shedding, and resource sharing among similar queries running in parallel, as summarized below.

18.1.5.1 Cost Metrics and Statistics

Traditional DBMSs use selectivity information and available indices to choose efficient query plans (e.g., those which require the fewest disk accesses). However, this cost metric does not apply to (possibly approximate) persistent queries, where processing cost per-unit-time is more appropriate [Kang et al., 2003]. Alternatively, if the stream arrival rates and output rates of query operators are known, then it may be possible to optimize for the highest output rate or to find a plan that takes the least time to output a given number of tuples [Tao et al., 2005; Urhan and Franklin, 2001; Viglas and Naughton, 2002]. Finally, quality-of-service metrics such as response time may also be used in DSMS query optimization [Abadi et al., 2003; Berthold et al., 2005; Schmidt et al., 2004, 2005].

18.1.5.2 Query Rewriting and Adaptive Query Optimization

Some of the DSMS query languages discussed in Section 18.1.2 introduce rewritings for new operators, e.g., selections and time-based sliding windows commute, but not selections and count-based windows [Arasu et al., 2006]. Other rewritings are similar to those used in relational databases, e.g., re-ordering a sequence of binary joins in order to minimize a particular cost metric. There has been some work in join ordering for data streams in the context of the rate-based model [Viglas and Naughton, 2002; Viglas et al., 2003]. Furthermore, adaptive re-ordering of pipelined stream filters [Babu et al., 2004a] and adaptive materialization of intermediate join results [Babu et al., 2005] have been investigated.

The notion of adaptivity is important in query rewriting; operators may need to be re-ordered on-the-fly in response to changes in system conditions. In particular, the cost of a query plan may change for three reasons: change in the processing time of an operator, change in the selectivity of a predicate, and change in the arrival rate of a stream [Adamic and Huberman, 2000]. Initial efforts on adaptive query plans include mid-query re-optimization [Kabra and DeWitt, 1998] and query scrambling, where the objective was to pre-empt any operators that become blocked and schedule other operators instead [Amsaleg et al., 1996b; Urhan et al., 1998b]. To further increase adaptivity, instead of maintaining a rigid tree-structured query plan, the Eddy approach [Adamic and Huberman, 2000] performs scheduling of each tuple separately by routing it through the operators that make up the query plan. In effect, the query plan is dynamically re-ordered to match current system conditions. This is accomplished by tuple routing policies that attempt to discover which operators are fast and selective, and those operators are scheduled first. A recent extension adds queue length as the third factor for tuple routing strategies in the presence of multiple distributed Eddies [Tian and DeWitt, 2003a]. There is, however, an important trade-off between the resulting adaptivity and the overhead required to route each tuple separately. More details on adaptive query processing may be found in [Babu and Bizarro, 2005; Babu and Widom, 2004; Gounaris et al., 2002a].

Adaptivity involves on-line reordering of a query plan and may therefore require that the internal state stored by some operators be migrated over to the new query plan consisting of a different arrangement of operators [Deshpande and Hellerstein, 2004; Zhu et al., 2004]. We do not discuss this issue further in this chapter.

18.1.6 Load Shedding and Approximation

The stream arrival rates may be so high that not all tuples can be processed, regardless of the (static or run-time) optimization techniques used. In this case, two types of load shedding may be applied—random or semantic—with the latter making use of stream properties or quality-of-service parameters to drop tuples believed to be less significant than others [Tatbul et al., 2003]. For an example of semantic load shedding, consider performing an approximate sliding window join with the objective

of attaining the maximum result size. The idea is that tuples that are about to expire or tuples that are not expected to produce many join results should be dropped (in case of memory limitations [Das et al., 2005; Li et al., 2006; Xie et al., 2005]), or inserted into the join state but ignored during the probing step (in case of CPU limitations [Ayad et al., 2006; Gedik et al., 2005; Han et al., 2006]). Note that other objectives are possible, such as obtaining a random sample of the join result [Srivastava and Widom, 2004].

In general, it is desirable to shed load in such a way as to minimize the drop in accuracy. This problem becomes more difficult when multiple queries with many operators are involved, as it must be decided where in the query plan the tuples should be dropped. Clearly, dropping tuples early in the plan is effective because all of the subsequent operators enjoy reduced load. However, this strategy may adversely affect the accuracy of many queries if parts of the plan are shared. On the other hand, load shedding later in the plan, after the shared sub-plans have been evaluated and the only remaining operators are specific to individual queries, may have little or no effect in reducing the overall system load.

One issue that arises in the context of load shedding and query plan generation is whether an optimal plan chosen without load shedding is still optimal if load shedding is used. It has been shown that this is indeed the case for sliding window aggregates, but not for queries involving sliding window joins [Ayad and Naughton, 2004].

Note that instead of dropping tuples during periods of high load, it is also possible to put them aside (e.g., spill to disk) and process them when the load has subsided [Liu et al., 2006; Reiss and Hellerstein, 2005]. Finally, note that in the case of periodic re-execution of persistent queries, increasing the re-execution interval may be thought of as a form of load shedding [Babcock et al., 2002; Cammert et al., 2006; Wu et al., 2005].

18.1.7 Multi-Query Optimization

As seen in Section 18.1.4.4, memory usage may be reduced by sharing internal data structures that store operator state [Denny and Franklin, 2005; Dobra et al., 2004; Zhang et al., 2005]. Additionally, in the context of complex queries containing stateful operators such as joins, computation may be shared by building a common query plan [Chen et al., 2000]. For example, queries belonging to the same group may share a plan, which produces the union of the results needed by the individual queries. A final selection is then applied to the shared result set and new answers are routed to the appropriate queries. An interesting trade-off appears between doing similar work multiple times and doing too much unnecessary work; techniques that balance this trade-off are presented in [Chen et al., 2002; Krishnamurthy et al., 2004; Wang et al., 2006]. For example, suppose that the workload includes several queries referencing a join of the same windows, but having a different selection predicate. If a shared query plan performs the join first and then routes the output to appropriate

queries, then too much work is being done because some of the joined tuples may not satisfy any selection predicate (unnecessary tuples are being generated). On the other hand, if each query performs its selection first and then joins the surviving tuples, then the join operator cannot be shared and the same tuples will be probed many times.

For selection queries, a possible multi-query optimization is to index the query predicates and store auxiliary information in each tuple that identifies which queries it satisfies [Chandrasekaran and Franklin, 2003; Demers et al., 2006; Hanson et al., 1999; Krishnamurthy et al., 2006; Lim et al., 2006; Madden et al., 2002a; Wu et al., 2004]. When a new tuple arrives for processing, its attribute values are extracted and matched against the query index to see which queries are satisfied by this tuple. Data and queries may be thought of as duals, in some cases reducing query processing to a multi-way join of the query predicate index and the data tables [Chandrasekaran and Franklin, 2003; Lim et al., 2006].

18.1.8 Stream Mining

In addition to querying as discussed in the previous sections, mining of stream data has been studied for a number of applications. Data mining involves the use of data analysis tools to discover previously unknown relationships and patterns in large data sets. The characteristics of data streams discussed above impose new challenges in performing mining tasks; many of the well-known techniques cannot be used. The major issues are the following:

- **Unbounded data set.** Traditional data mining algorithms are based on the assumption that they can access the full data set. However, this is not possible in data streams, where only a portion of the old data is available and much of the old data are discarded. Hence, data mining techniques that require multiple scan over the entire data set cannot be used.
- **"Messy" data.** Data are never entirely clean, but in traditional data mining applications, they can be cleaned before the application is run. In many stream applications, due to the high arrival rates of data streams, this is not always possible. Given that in many cases the data that are read from sensors and other sources of stream data are already quite noisy, the problem is even more serious.
- **Real-time processing.** Data mining over traditional data is typically a batch process. Although there are obvious efficiency concerns in analyzing these data, they are not as severe as those on data streams. Since data arrival is continuous and potentially at high rate, the mining algorithms have to have real-time performance.
- **Data evolution.** As noted earlier traditional data sets can be assumed to be static, i.e., the data is a sample from a static distribution. However, this is not true for many real-world data streams, since they are generated over long

periods of time during which the underlying phenomena can change resulting in significant changes in the distribution of the data values. This means that some mining results that were previously generated may no longer be valid. Therefore, a data stream mining technique must have the ability to detect changes in the stream, and to automatically modify its mining strategy for different distributions.

In the remainder, we will summarize some stream mining techniques. We divide the discussion into two groups: general processing techniques, and specific data mining tasks and their algorithms [Gaber et al., 2005]. Data processing techniques are general approaches to process the stream data before specific tasks can be applied. These consist of the following:

Sampling. As discussed earlier, data stream sampling is the process of choosing a suitable representative subset from the stream of interest. In addition to the major use of stream sampling to reduce the potentially infinite size of the stream to a bounded set of samples, it can be utilized to clean "messy" data and to preserve representative sets for the historical distributions. However, since some data elements of the stream are not looked at, in general, it is impossible to guarantee that the results produced by the mining application using the samples will be identical to the results returned on the complete stream up to the most recent time. Therefore, one of the most critical tasks for stream sampling techniques is to provide guarantees about how much the results obtained using the samples differ from the non-sampling based results.

Load shedding. The arrival speed of elements in data streams are usually unstable, and many data stream sources are prone to dramatic spikes in load. Therefore, stream mining applications must cope with the effects of system overload. Maximizing the mining benefits under resource constraints is a challenging task. Load shedding techniques as discussed earlier are helpful.

Synopsis maintenance. Synopsis maintenance processes create synopses or "sketches" for summarizing the streams and were introduced earlier in this chapter. A synopsis does not represent all characteristics of a stream, but rather some "key features" that might be useful for tuning the stream mining processes and further analyzing the streams. It is especially useful for stream mining applications that are expecting various streams as input, or an input stream with frequent distribution changes. When the stream changes, some re-computation, either from scratch or incrementally, has to be done. An efficient synopsis maintenance process can generate summary of the stream shortly after the change, and the stream mining application can re-adjust its settings or switch to another mining technique based on these precious information.

Change detection. When the distribution of the stream changes, previous mining results may no longer be valid under the new distribution, and the mining technique must be adjusted to maintain good performance for the new distribution. Hence, it is critical for the distribution changes in a stream to be detected in real-time so that the stream mining application can react promptly.

There are basically two different tracks oftechniques for detecting changes. One track is to look at the natureof the dataset and determine if that set has evolved [Kifer et al., 2004; Aggarwal, 2003, 2005], and the othertrack is to detect if an existing data model is no longer suitablefor recent data, which implies the concept drifting [Hulten et al., 2001; Wang et al., 2003a; Fan, 2004; Gama et al., 2005]belong to the second track.

Now we take a look at some of the popular stream mining tasks and how they can be accomplished in this environment. We focus on clustering, classification, frequency counting and association rule mining, and time series analysis.

Clustering. Clustering groups together data with similar behavior. It can be thought of as partitioning or segmenting elements into groups (clusters) that may or may not be disjoint. In many cases, the answer to a clustering problem is not unique, i.e., many answers can be found, and interpreting the practical meaning of each cluster may be difficult.

Aggarwal et al. [2003] have proposed a framework for clustering data streams that uses an online component to store summarized information about the streams, and an offline component that performs clustering on the summarized data. This framework has been extended in HPStream in a way that can find projected clusters for high dimensional data streams [Aggarwal et al., 2004] .

The existing clustering algorithms can be categorized into decision tree based ones (e.g., [Domingos and Hulten, 2000; Gama et al., 2005; Hulten et al., 2001; Tao and Özsu, 2009]) and k-mean (or k-median) based approaches (e.g., [Babcock et al., 2002; Charikar et al., 1997, 2003; Guha et al., 2003; Ordonez, 2003]).

Classification. Classification maps data into predefined groups (classes). Its difference from clustering is that, in classification, the number of groups is predetermined and fixed. Similar to clustering, classification techniques can also adopt the decision tree model (e.g., [Ding et al., 2002; Ganti et al., 2002]). Two decision tree classifiers — Interval Classifier [Agrawal et al., 1992] and SPRINT [Shafer et al., 1996] — can mine databases that do not fit in main memory, and are thus are suitable for data streams. The VFDT [Domingos and Hulten, 2000] and CVFDT [Hulten et al., 2001] systems originally designed for stream clustering can also be adopted for classification tasks.

Frequency counting and association rule mining. The problem of frequency counting, and mining association rules (frequent itemsets) has long been recognized as an important issue. However, although mining frequent itemsets has been widely studied in data mining and a number of efficient algoirthms exist, extending these to data streams is challenging, especially for streams with non-static distributions [Jiang and Gruenwald, 2006].

Mining frequent itemsets is a continuous process that runs throughout a stream's life span. Since the total number of itemsets is exponential, making it impractical to keep count of each itemset in order to incrementally adjust the frequent itemsets as new data items arrive. Usually only the itemsets that are already known to be frequent are recorded and monitored, and counters of infrequent itemsets are discarded [Chakrabarti et al., 2002; Cormode and Muthukrishnan, 2003; Demaine

et al., 2002; Halatchev and Gruenwald, 2005] . However, since data streams can change over time, an itemset that was once infrequent may become frequent if the distribution changes. Such (new) frequent itemsets are difficult to detect, since mining data streams is a one-pass procedure and history information is not retrievable.

Time series analysis. In general, a time series is a set of attribute values over a period of time. Usually a time series consists of only numeric values, either continuous or discrete. Consequently, it is possible to model data streams that contain only numeric values as time series. This allows one to use analysis techniques that have been developed on time series for some types of stream data. Mining tasks over time series can be briefly classified into two types: pattern detection and trend analysis. A typical mining task for pattern detection is the following: given a sample pattern or a base time series with a certain pattern, find all the time series that contain this pattern. The tasks for trend prediction are detecting trends in time series and predicting the upcoming trends.

18.2 Cloud Data Management

Cloud computing is the latest trend in distributed computing and has been the subject of much hype. The vision encompasses on demand, reliable services provided over the Internet (typically represented as a cloud) with easy access to virtually infinite computing, storage and networking resources. Through very simple web interfaces and at small incremental cost, users can outsource complex tasks, such as data storage, system administration, or application deployment, to very large data centers operated by cloud providers. Thus, the complexity of managing the software/hardware infrastructure gets shifted from the users' organization to the cloud provider.

Cloud computing is a natural evolution, and combination, of different computing models proposed for supporting applications over the web: service oriented architectures (SOA) for high-level communication of applications through web services, utility computing for packaging computing and storage resources as services, cluster and virtualization technologies to manage lots of computing and storage resources, autonomous computing to enable self-management of complex infrastructure, and grid computing to deal with distributed resources over the network. However, what makes cloud computing unique is its ability to provide various levels of functionality such as infrastructure, platform, and application as services that can be combined to best fit the users' requirements [Cusumano, 2010]. From a technical point of view, the grand challenge is to support in a cost-effective way, the very large scale of the infrastructure that has to manage lots of users and resources with high quality of service.

Cloud computing has been developed by web industry giants, such as Amazon, Google, Microsoft and Yahoo, to create a new, huge market. Virtually all computer industry players are interested in cloud computing. Cloud providers have developed

new, proprietary technologies (e.g., Google File System), typically with specific, simple applications in mind. There are already open source implementations (e.g., Hadoop Distributed File System) with much contribution from the research community. As the need to support more complex applications increases, the interest of the research community is steadily growing. In particular, data management in cloud computing is becoming a major research direction which we think can capitalize on distributed and parallel database techniques.

The rest of this section is organized as follows. First, we give a general taxonomy of the different kinds of clouds, and a discussion of the advantages and potential disadvantages. Second, we give an overview of grid computing, with which cloud computing is sometimes confused, and point out the main differences. Third, we present the main cloud architectures and associated functions. Fourth, we present the current solutions for data management in the cloud, in particular, data storage, database management and parallel data processing. Finally, we discuss open issues in cloud data management.

18.2.1 Taxonomy of Clouds

In this section, we first give a definition of cloud computing, with the main categories of cloud services. Then, we discuss the main data-intensive applications that are suitable for the cloud and the main issues, in particular, security.

Agreeing on a precise definition of cloud computing is difficult as there are many different perspectives (business, market, technical, research, etc.). However, a good working definition is that a "cloud provides on demand resources and services over the Internet, usually at the scale and with the reliability of a data center" [Grossman and Gu, 2009]. This definition captures well the main objective (providing on-demand resources and services over the Internet) and the main requirements for supporting them (at the scale and with the reliability of a data center).

Since the resources are accessed through services, everything gets delivered as a service. Thus, as in the services industry, this enables cloud providers to propose a pay-as-you-go pricing model, whereby users only pay for the resources they consume. However, implementing a pricing model is complex as users should be charged based on the level of service actually delivered, e.g., in terms of service availability or performance. To govern the use of services by customers and support pricing, cloud providers use the concept of Service Level Agreement (SLA), which is critical in the services industry (e.g., in telecoms), but in a rather simple way. The SLA (between the cloud provider and any customer) typically specifies the responsabilities, guarantees and service commitment. For instance, the service commitment might state that the service uptime during a billing cycle (e.g., a month) should be at least 99%, and if the commitment is not met, the customer should get a service credit.

Cloud services can be divided in three broad categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS).

- **Infrastructure-as-a-Service (IaaS).** IaaS is the delivery of a computing infrastructure (i.e., computing, networking and storage resources) as a service. It enables customers to scale up (add more resources) or scale down (release resources) as needed (and only pay for the resources consumed). This important capability is called *elasticity* and is typically achieved through *server virtualization*, a technology that enables multiple applications to run on the same physical server as virtual machines, i.e., as if they would run on distinct physical servers. Customers can then requisition computing instances as virtual machines and add and attach storage as needed. An example of popular IaaS is Amazon web Services.
- **Software-as-a-Service (SaaS).** SaaS is the delivery of application software as a service. It generalizes the earlier Application Service Provider (ASP) model whereby the hosted application is fully owned, operated and maintained by the ASP. With SaaS, the cloud provider allows the customer to use hosted applications (as with ASP) but also provides tools to integrate other applications, from different vendors or even developed by the customer (using the cloud platform). Hosted applications can range from simple ones such as email and calendar to complex applications such as customer relationship management (CRM), data analysis or even social networks. An example of popular SaaS is Safesforce CRM system.
- **Platform-as-a-Service (PaaS).** PaaS is the delivery of a computing platform with development tools and APIs as a service. It enables developers to create and deploy custom applications directly on the cloud infrastructure, in virtual machines, and integrate them with applications provided as SaaS. An example of popular PaaS is Google Apps.

By using a combination of IaaS, SaaS and PaaS, customers could move all or part of their information technology (IT) services to the cloud, with the following main benefits:

- **Cost.** The cost for the customer can be greatly reduced since the IT infrastructure does not need to be owned and managed; billing is only based only on resource consumption. For the cloud provider, using a consolidated infrastructure and sharing costs for multiple customers reduces the cost of ownership and operation.
- **Ease of access and use.** The cloud hides the complexity of the IT infrastructure and makes location and distribution transparent. Thus, customers can have access to IT services anytime, and from anywhere with an Internet connection.
- **Quality of Service (QoS).** The operation of the IT infrastructure by a specialized provider that has extensive experience in running very large infrastructures (including its own infrastructure) increases QoS.
- **Elasticity.** The ability to scale resources out, up and down dynamically to accommodate changing conditions is a major advantage. In particular, it makes it easy for customers to deal with sudden increases in loads by simply creating more virtual machines.

However, not all corporate applications are good candidates for being "cloudified" [Abadi, 2009]. To simplify, we can classify corporate applications between the two

main classes of data-intensive applications which we already discussed: OLTP and OLAP. Let us recall their main characteristics. OLTP deals with operational databases of average sizes (up to a few terabytes), that are write-intensive, and require complete ACID transactional properties, strong data protection and response time guarantees. On the other hand, OLAP deals with historical databases of very large sizes (up to petabytes), that are read-intensive, and thus can accept relaxed ACID properties. Furthermore, since OLAP data are typically extracted from operational OLTP databases, sensitive data can be simply hidden for analysis (e.g., using anonymization) so that data protection is not as crucial as in OLTP.

OLAP is more suitable than OLTP for cloud primarily because of two cloud characteristics (see the detailed discussion in [Abadi, 2009]): elasticity and security. To support elasticity in a cost-effective way, the best solution, which most cloud providers adopt, is a shared-nothing cluster architecture. Recall from Section 14.1 that shared-nothing provides high-scalability but requires careful data partitioning. Since OLAP databases are very large and mostly read-only, data partitioning and parallel query processing are effective. However, it is much harder to support OLTP on shared-nothing because of ACID guarantees, which require complex concurrency control. For these reasons and because OLTP databases are not so large, shared-disk is the preferred architecture for OLTP. The second reason that OLTP is not so suitable for cloud is that the corporate data get stored at an untrusted host (the provider site). Storing corporate data at an untrusted third-party, even with a carefully negotiated SLA with a reliable provider, creates resistance from some customers because of security issues. However, this resistance is much reduced for historical data, and with anonymized sensitive data.

There are currently two main solutions to address the security issue in clouds: internal cloud and virtual private cloud. The mainstream cloud approach is generally called *public cloud*, because the cloud is available to anyone on the Internet. An *internal cloud* (or *private cloud*) is the use of cloud technologies for managing a company's data center, but in a private network behind a firewall. This brings much tighter security and many of the advantages of cloud computing. However, the cost advantage tends to be much reduced because the infrastructure is not shared with other customers. Nonetheless, an attractive compromise is the *hybrid cloud* which connects the internal cloud (e.g., for OLTP) with one or more public clouds (e.g., for OLAP). As an alternative to internal clouds, cloud providers such as Amazon and Google have proposed *virtual private clouds* with the promise of a similar level of security as an internal cloud, but within a public cloud. A virtual private cloud provides a Virtual Private Network (VPN) with security services to the customers. Virtual private clouds can also be used to develop hybrid clouds, with tighter security integration with the internal cloud.

One earlier criticism of cloud computing is that customers get locked in proprietary clouds. It is true that most clouds are proprietary and there are no standards for cloud interoperability. But this is changing with open source cloud software such as Hadoop, an Apache project implementing Google's major cloud services such as Google File System and MapReduce, and Eucalyptus, an open source cloud software infrastructure, which are attracting much interest from research and industry.

18.2.2 Grid Computing

Like cloud computing, grid computing enables access to very large compute and storage resources over the web. It has been the subject of much research and development over the last decade. Cloud computing is somewhat more recent and there are similarities but also differences between the two computing models. In this section, we discuss the main aspects of grid computing and end with a comparison with cloud computing.

Grid computing has been initially developed for the scientific community as a generalization of cluster computing, typically to solve very large problems (that require a lot of computing power and/or access to large amounts of data) using many computers over the web. Grid computing has also gained some interest in enterprise information systems. For instance, IBM and Oracle (since Oracle 10g with g standing for grid) have been promoting grid computing with tools and services for both scientific and enterprise applications.

Grid computing enables the virtualization of distributed, heterogeneous resources using web services [Atkinson et al., 2005]. These resources can be data sources (files, databases, web sites, etc.), computing resources (multiprocessors, supercomputers, clusters) and application resources (scientific applications, information management services, etc.). Unlike the web, which is client-server oriented, the grid is demand-oriented: users send requests to the grid which allocates them to the most appropriate resources to handle them. A grid is also an organized, secured environment managed and controlled by administrators. An important unit of control in a grid is the Virtual Organization (VO), i.e., a group of individuals, organizations or companies that share the same resources, with common rules and access rights. A grid can have one or more VOs, and may have different size, duration and goal.

Compared with cluster computing, which only deals with parallelism, the grid is characterized with high heterogeneity, large-scale distribution and large-scale parallelism. Thus, it can offer advanced services on top of very large amounts of distributed data.

Depending on the contributed resources and the targeted applications, many different kinds of grids and architectures are possible. The earlier computational grids typically aggregate very powerful sites (supercomputers, clusters) to provide high-performance computing for scientific applications (e.g., physics, astronomy). Data grids aggregate heterogeneous data sources (like a distributed database) and provide additional services for data discovery, delivery and use to scientific applications. More recently, enterprise grids [Jiménez-Peris et al., 2007] have been proposed to aggregate information system resources, such as web servers, application servers and database servers, in the enterprise.

Figure 18.4 illustrates a typical grid scenario, inspired by the Grid5000 platform in France, with two computing sites (clusters 1 and 2) and one storage site (cluster 3) accessible to authorized users. Each site has one cluster with service nodes and either compute or storage nodes. Service nodes provide common services for users (access, resource reservation, deployment) and administrators (infrastructure services) and are available at each site, through the replication of directories and catalogs.

Compute nodes provide the main computing power while storage nodes provide storage capacity (i.e., lots of disks). The basic communication between grid sites (e.g., to deploy an application or a system image) is through web services (WS) calls (to be discussed shortly). But for distributing computation between compute nodes at two different sites, communication is typically through the standard Message Passing Interface (MPI).

A typical scenario for solving a large scientific problem P is the following. P is initially decomposed (by a scientist programmer User 1) into two subproblems P_1 and P_2, each being solved through a parallel program to be run at one computing site. If P_1 and P_2 are independent then there is no need for communication between the computing sites. If there are computing dependencies, e.g., P_2 consumes results of P_1, communication between P_1 and P_2 must be specified and implemented through MPI. The data produced by P_1 and P_2 could then be sent to the storage site, typically using WS calls. To run P on the grid, a user must first reserve the computing resources (e.g., a needed number of cluster nodes at site 1 and 2) and storage resources (at site 3), deploy the jobs corresponding to the programs, and then start their parallel executions at site 1 and 2, which will produce data and send them to site 3. The resource allocation and the scheduling of job executions at the clusters are done by the grid middleware in a way that guarantees fair access to the reserved resources. More complex scenarios can also involve the distributed execution of workflows. On the other hand, User 2 can simply reserve storage capacity and use it for saving her local data (using the store interface).

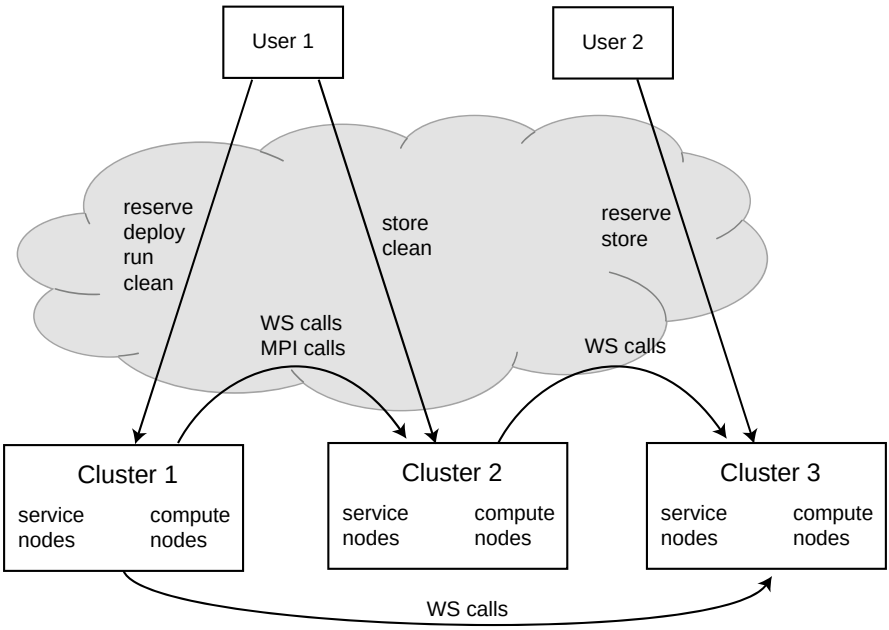


Fig. 18.4 A Grid Scenario

A common need of different kinds of grids is interoperability of heterogeneous resources. To address this need, the Globus Alliance, which represents the grid community, has defined the Open Grid Services Architecture (OGSA) as a standard SOA and a framework to create grid solutions using WS standards. OGSA provides three main layers to build grid applications: (1) resources layer, (2) web services layer, and (3) high-level grid services layer. The first layer provides an abstraction of the physical resources (servers, storage, network) that are managed by logical resources such as database systems, file systems, or workflow managers, all encapsulated by WS. The second layer extends WS, which are typically stateless, to deal with stateful grid services, i.e., those that can retain data between multiple invocations. This capability is useful for instance to access a resources state, e.g., the load of a server, through WS. Stateful grid services can be created and destroyed (using a grid service factory), and have an internal state which can be observed or even changed after notifications from other grid services. The third layer provides high-level grid-specific services such as resource provisioning, data management, security, workflow, and monitoring to ease the development and management of grid applications.

The adoption of WS in enterprise information systems has made OGSA appealing and several offerings for enterprise grids are based on the Globus platform (e.g., Oracle 11g). Web service standards are useful for grid data management: XML for data exchange, XMLSchema for schema description, Simple Object Access Protocol (SOAP) for remote procedure calls, UDDI for directory access, Web Service Definition Language (WSDL) for data source description, WS-Transaction for distributed transactions, Business Process Execution Language (BPEL) for workflow control, etc.

The main solutions for grid data management, in the context of computational grids, are file-based [Pacitti et al., 2007b]. A basic solution, used in Globus, is to combine global directory services to locate files and a secure file transfer protocol. Although simple, this solution does not provide distribution transparency as it requires the application to explicitly transfer files. Another solution is to use a distributed file system for the grid that can provide location-independent file access and transparent replication [Zhang and Honeyman, 2008].

Recent solutions have recognized the need for high-level data access and extended the distributed database architecture whereby clients send database requests to a grid multidatabase server that forwards them transparently to the appropriate database servers. These solutions rely on some form of global directory management, where directories can be distributed and replicated. In particular, users are able to use a high-level query language (SQL) to describe the desired data as with OGSA-DAI (OGSA Database Access and Integration), an OGSA standard for accessing and integrating distributed data [Antonioletti et al., 2005]. OGSA-DAI is a popular multidatabase system that provides uniform access to heterogeneous data sources (e.g., relational databases, XML databases or files) via WS within grids. Its architecture is similar to the mediator/wrapper architecture described in Chapters 1and 9 with the wrappers implemented by WS. The OGSA-DAI mediator includes a distributed query processor which automatically transforms a multidatabase query into a distributed QEP that specifies the WS calls to get the required data from each database wrapper.

We end this section with a discussion of the advantages and disadvantages of grid computing. The main advantages come from the distributed architecture when it uses clusters at each site, as it provides scalability, performance (through parallelism) and availability (through replication). It is also a cost-effective alternative to a huge supercomputer to solve larger, more complex problems in a shorter time. Another advantage is that existing resources are better used and shared with other organizations. The main disadvantages also come from the highly distributed architecture, which is complex for both administrators and developers. In particular, sharing resources across administrative domains is a political challenge for participating organizations as it is hard to assess their cost/benefits.

Compared with cloud computing, there are important differences in terms of objectives and architecture. Grid computing fosters collaboration among participating organizations to leverage existing resources whereas cloud computing provides a rather fixed (distributed) infrastructure to all kinds of users (and customers). Thus, SLA and pay-per-use are essential in cloud computing. The grid architecture is potentially much more distributed than the cloud architecture that typically consists of a few sites in different geographical regions, but each site being a very huge data center. Therefore, the scalability issue at a site (in terms of numbers of users or numbers of server nodes) is much harder in cloud computing. Finally, a major difference is that there are no standards such as OGSA for cloud interoperability.

18.2.3 Cloud architectures

Unlike in grid computing, there is no standard cloud architecture and there will probably never be one, since different cloud providers will provide different cloud services (IaaS, PaaS, SaaS) in different ways (public, private, virtual private, ...) depending on their business models. Thus, in this section, we discuss the main cloud architectures in order to identify the underlying technologies and functions. This is useful to be able to focus on data management (in the next section).

Figure 18.5 illustrates a typical cloud scenario, inspired by that of a popular IaaS/PaaS provider. This scenario is also useful for comparison with the typical grid scenario in Figure 18.4. We assume one cloud provider with two sites, each with the same capabilities and cluster architecture. Thus, any user can access any site to get the needed service as if there were only one site, so the cloud appears “centralized”. This is one major difference with grid as distribution can be completely hidden. However, distribution happens under the cover, e.g., to replicate data automatically from one site to the other in order to resist to site failure. Then, to solve the large scientific problem P, User 1 now does not need to decompose it into two subproblems, but she does need to provide a parallel version of P to be run at Site 1. This is done by creating a *virtual machine* (VM) (sometimes called *computing instance*) with executable application code and data, then starting as many VMs as needed for the parallel execution and finally terminating. User 1 is then charged only for the resources (VMs) consumed. The allocation of VMs to physical machines at Site 1 is

done by the cloud middleware in a way that optimizes global resource consumption while satisfying the SLA. On the other hand, similar to the grid scenario, User 2 can also reserve storage capacity and use it for saving her local data.

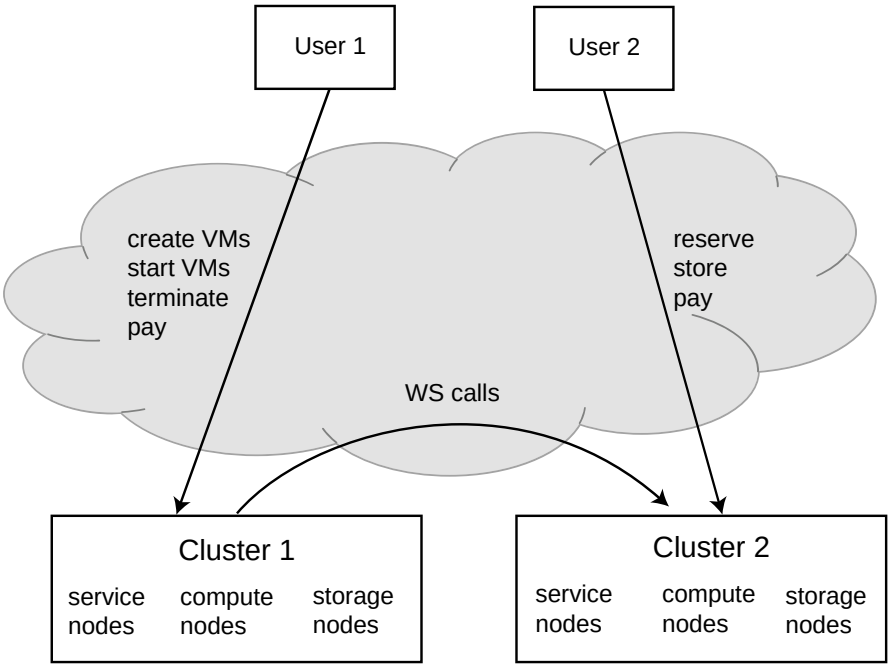


Fig. 18.5 A Cloud Scenario

We can distinguish the cloud architectures between infrastructure (IaaS) and software/platform (SaaS/PaaS). All architectures can be supported by a network of shared-nothing clusters. For IaaS, the preferred architectural model derives from the need to provide computing instances on demand. To support computing instances on demand, as in the scenario in Figure 18.5, the main solution is to rely on server virtualization, which enables VMs to be provisioned and decommissioned as needed. Server virtualization can be well supported by a shared-nothing cluster architecture. For SaaS/PaaS, many different architectural models can be used depending on the targeted services and applications. For instance, to support enterprise applications, a typical architecture is n-tier with web servers, application servers, database servers and storage servers, all organized in a cluster architecture. Server virtualization can also be used in such architecture. For data storage virtualization, SAN can be used to provide shared-disk access to service or compute nodes. As for grids, communication between applications and services is typically done through WS or message passing.

The main functions provided by clouds are similar to those found in grids: security, directory management, resource management (provisioning, allocation, monitoring) and data management (storage, file management, database management, data

replication). In addition, clouds provide support for pricing, accounting and SLA management.

18.2.4 Data management in the cloud

For managing data, cloud providers could rely on relational DBMS technology, all of which have distributed and parallel versions. However, relational DBMSs have been lately criticized for their "one size fits all" approach [Stonebraker, 2010]. Although they have been able to integrate support for all kinds of data (e.g., multimedia objects, XML documents) and new functions, this has resulted in a loss of performance, simplicity and flexibility for applications with specific, tight performance requirements. Therefore, it has been argued that more specialized DBMS engines are needed. For instance, column-oriented DBMSs [Abadi et al., 2008], which store column data together rather than rows in traditional row-oriented relational DBMSs, have been shown to perform more than an order of magnitude better on OLAP workloads. Similarly, as discussed in Section 18.1, DSMSs are specifically architected to deal efficiently with data streams which traditional DBMS cannot even support.

The "one size does not fit all" argument generally applies to cloud data management as well. However, internal clouds or virtual private clouds for enterprise information systems, in particular for OLTP, may use traditional relational DBMS technology. On the other hand, for OLAP workloads and web-based applications on the cloud, relational DBMS provide both too much (e.g., ACID transactions, complex query language, lots of tuning parameters), and too little (e.g., specific optimizations for OLAP, flexible programming model, flexible schema, scalability) [Ramakrishnan, 2009]. Some important characteristics of cloud data have been considered for designing data management solutions. Cloud data can be very large (e.g., text-based or scientific applications), unstructured or semi-structured, and typically append-only (with rare updates). And cloud users and application developers may be in high numbers, but not DBMS experts. Therefore, current cloud data management solutions have traded consistency for scalability, simplicity and flexibility.

In this section, we illustrate cloud data management with representative solutions for distributed file management, distributed database management and parallel database programming.

18.2.4.1 Distributed File Management

The Google File System (GFS) [Ghemawat et al., 2003] is a popular distributed file system developed by Google for its internal use. It is used by many Google applications and systems, such as Bigtable and MapReduce, which we discuss next. There are also open source implementations of GFS, such as Hadoop Distributed File System (HDFS), a popular Java product.

Similar to other distributed file systems, GFS aims at providing performance, scalability, fault-tolerance and availability. However, the targeted systems, shared-nothing clusters, are challenging as they are made of many (e.g., thousands of) servers built from inexpensive hardware. Thus, the probability that any server fails at a given time is high, which makes fault-tolerance difficult. GFS addresses this problem. It is also optimized for Google data-intensive applications, such as search engine or data analysis. These applications have the following characteristics. First, their files are very large, typically several gigabytes, containing many objects such as web documents. Second, workloads consist mainly of read and append operations, while random updates are rare. Read operations consist of large reads of bulk data (e.g., 1 MB) and small random reads (e.g., a few KBs). The append operations are also large and there may be many concurrent clients that append the same file. Third, because workloads consist mainly of large read and append operations, high throughput is more important than low latency.

GFS organizes files as a tree of directories and identifies them by pathnames. It provides a file system interface with traditional file operations (create, open, read, write, close, and delete file) and two additional operations: snapshot and record append. Snapshot allows creating a copy of a file or of a directory tree. Record append allows appending data (the record) to a file by concurrent clients in an efficient way. A record is appended atomically, i.e., as a continuous byte string, at a byte location determined by GFS. This avoids the need for distributed lock management that would be necessary with the traditional write operation (which could be used to append data).

The architecture of GFS is illustrated in Figure 18.6. Files are divided into fixed-size partitions, called *chunks*, of large size, i.e., 64 MB. The cluster nodes consist of GFS clients that provide the GFS interface to applications, chunk servers that store chunks and a single GFS master that maintains file metadata such as namespace, access control information, and chunk placement information. Each chunk has a unique id assigned by the master at creation time and, for reliability reasons, is replicated on at least three chunk servers (in Linux files). To access chunk data, a client must first ask the master for the chunk locations, needed to answer the application file access. Then, using the information returned by the master, the client can request the chunk data to one of the replicas.

This architecture using single master is simple. And since the master is mostly used for locating chunks and does not hold chunk data, it is not a bottleneck. Furthermore, there is no data caching at either clients or chunk servers, since it would not benefit large reads. Another simplification is a relaxed consistency model for concurrent writes and record appends. Thus, the applications must deal with relaxed consistency using techniques such as checkpointing and writing self-validating records. Finally, to keep the system highly available in the face of frequent node failures, GFS relies on fast recovery and replication strategies.

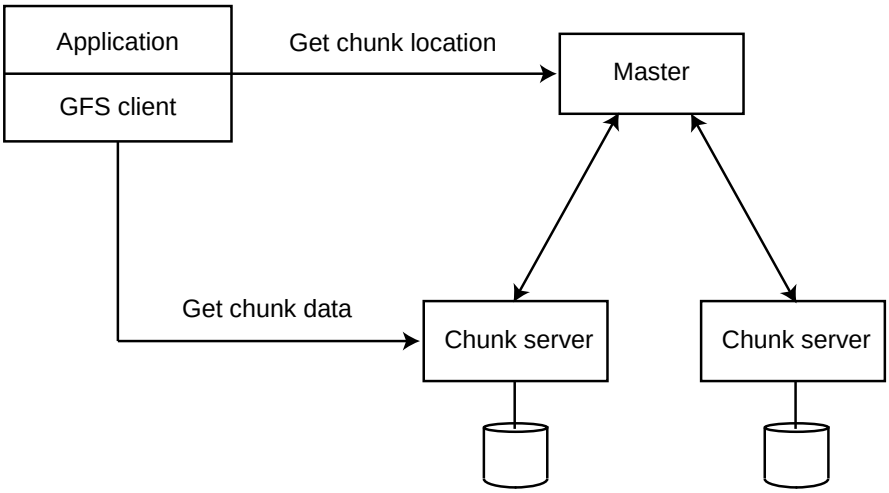


Fig. 18.6 GFS Architecture

18.2.4.2 Distributed Database Management

We can distinguish between two kinds of solutions: online distributed database services and distributed database systems for cloud applications. Online distributed database services such as Amazon SimpleDB and Google Base enable any web user to add and manipulate structured data in a database in a very simple way, without having to define a schema. For instance, SimpleDB provides basic database functionality including scan, filter, join and aggregate operators, caching, replication and transactions, but no complex operators (e.g., union), no query optimizer and no fault-tolerance. Data are structured as (attribute name, value) pairs, all automatically indexed so there is no need for administration. Google Base is a simpler online database service (as a Beta version at the time of this writing) which enables a user to add and retrieve structured data through predefined forms, with predefined attributes (e.g., ingredient for a recipe), thus avoiding the need for schema definition. Data in Google Base can then be searched through other tools, such as the web search engine.

Distributed database systems for cloud applications emphasize scalability, fault-tolerance and availability, sometimes at the expense of consistency or ease of development. We illustrate this approach with two popular solutions: Google Bigtable and Yahoo! PNUTS.

Bigtable.

Bigtable is a database storage system for a shared-nothing cluster [Chang et al., 2008]. It uses GFS for storing structured data in distributed files, which provides

fault-tolerance and availability. It also uses a form of dynamic data partitioning for scalability. And like GFS, it is used by popular Google applications, such as Google Earth, Google Analytics and Orkut. There are also open source implementations of Bigtable, such as Hadoop Hbase, which runs on HDFS.

Bigtable supports a simple data model that resembles the relational model, with multi-valued, timestamped attributes. We briefly describe this model as it is the basis for Bigtable implementation that combines aspects of row-store and column-store DBMS. We use the terminology of the original proposal [Chang et al., 2008], in particular, the basic terms "row" and "column" (instead of tuple and attribute). However, for consistency with the concepts we have used so far, we present the Bigtable data model as a slightly extended relational model[1]. Each row in a table (or Bigtable) is uniquely identified by a *row key*, which is an arbitrary string (of up to 64KB in the original system). Thus, a row key is like a mono-attribute key in a relation. A more original concept is that of a *column family* which is a set of columns (of the same type), each identified by a *column key*. A column family is a unit of access control and compression. The syntax for naming column keys is `family:qualifier`. The column family name is like a relation attribute name. The qualifier is like a relation attribute value, but used as a name as part of the column key to represent a single data item. This allows the equivalent of multi-valued attributes within a relation, but with the capability of naming attribute values. In addition, the data identified by a column key within a row can have multiple versions, each identified by a timestamp (a 64 bit integer).

Figure 18.7 shows an example a row in a Bigtable, as a relational style representation of the example [Chang et al., 2008]. The row key is a reverse URL. The Contents:column family has only one column key that represents the web page contents, with two versions (at timestamps t_1 and t_5). The Language:family has also only one column key that represents the web page language, with one version. The Anchor: column family has two column keys, i.e., Anchor:inria.fr and Anchor:uwaterloo.ca, which represent two anchors. The anchor source site name (e.g., inria.fr) is used as qualifier and the link text as value.

Bigtable provides a basic API for defining and manipulating tables, within a programming language such as C++. The API offers various operators to write and update values, and to iterate over subsets of data, produced by a scan operator. There are various ways to restrict the rows, columns and timestamps produced by a scan, as in a relational select operator. However, there are no complex operators such as join or union, which need to be programmed using the scan operator. Transactional atomicity is supported for single row updates only.

To store a table in GFS, Bigtable uses range partitioning on the row key. Each table is divided into partitions called *tablets*, each corresponding to a row range. Partitioning is dynamic, starting with one tablet (the entire table range) that is subsequently split into multiple tablets as the table grows. To locate the (user) tablets in GFS, Bigtable uses a metadata table, which is itself partitioned in metadata tablets, with a single root tablet stored at a master server, similar to GFSs master. In addition

[1] In the original proposal, a Bigtable is defined as a multidimensional map, indexed by a row key, a column key and a timestamp, each cell of the map being a single value (a string).

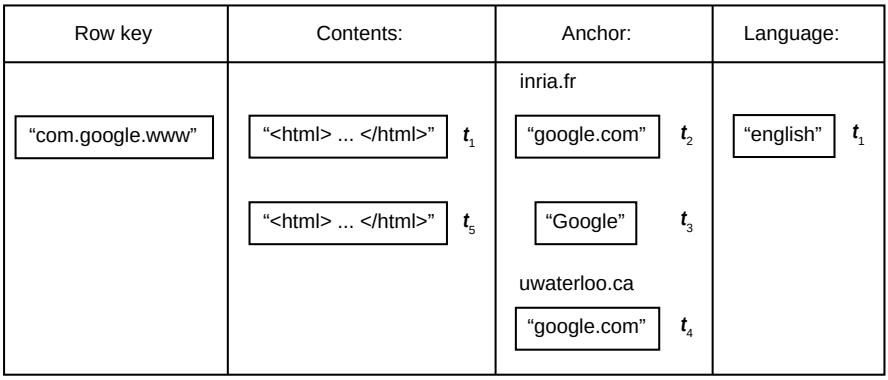


Fig. 18.7 Example of a Bigtable Row

to exploiting GFS for scalability and availability, Bigtable uses various techniques to optimize data access and minimize the number of disk accesses, such as compression of column families, grouping of column families with high locality of access and aggressive caching of metadata information by clients.

PNUTS.

PNUTS is a parallel and distributed database system for Yahoo!'s cloud applications [Cooper et al., 2008]. It is designed to serve web applications, which typically do not need complex queries, but require good response time, scalability and high availability and can tolerate relaxed consistency guarantees for replicated data. PNUTS is used internally at Yahoo! for various applications such as user database, social networks, content metadata management and shopping listings management.

PNUTS supports the basic relational data model, with tables of flat records. However, arbitrary structures are allowed within attributes of Binary Long Object (Blob) type. Schemas are flexible as new attributes can be added at any time even though the table is being queried or updated, and records need not have values for all attributes. PNUTS provides a simple query language with selection and projection on a single relation. Updates and deletes must specify the primary key.

PNUTS provides a replica consistency model that is between strong consistency and eventual consistency (see Chapter 13 for detailed definitions). This model is motivated by the fact that web applications typically manipulate only one record at a time, but different records may be used under different geographic locations. Thus, PNUTS proposes *per-record timeline consistency*, which guarantees that all replicas of a given record apply all updates to the record in the same order. Using this consistency model, PNUTS supports several API operations with different guarantees. For instance, **Read-any** returns a possibly stale version of the record; **Read-latest** returns the latest copy of the record; **Write** performs a single atomic write operation.

Database tables are horizontally partitioned into tablets, through either range partitioning or hashing, which are distributed across many servers in a cluster (at a site). Furthermore, sites in different geographical regions maintain a complete copy of the system and of each table. An original aspect is the use of a publish/subscribe mechanism, with guaranteed delivery, for both reliability and replication. This avoids the need to keep a traditional database log as the publish/subscribe mechanism is used to replay lost updates.

18.2.4.3 Parallel Data Processing

We illustrate parallel data processing in the cloud with MapReduce, a popular programming framework for processing and generating large datasets [Dean and Ghemawat, 2004]. MapReduce was initially developed by Google as a proprietary product to process large amounts of unstructured or semi-structured data, such as web documents and logs of web page requests, on large shared-nothing clusters of commodity nodes and produce various kinds of data such as inverted indices or URL access frequencies. Different implementations of MapReduce are now available such as Amazon MapReduce (as a cloud service) or Hadoop MapReduce (as open source software).

MapReduce enables programmers to express in a simple, functional style their computations on large data sets and hides the details of parallel data processing, load balancing and fault-tolerance. The programming model includes only two operations, *map* and *reduce*, which we can find in many functional programming languages such as Lisp and ML. The Map operation is applied to each record in the input data set to compute one or more intermediate (key,value) pairs. The Reduce operation is applied to all the values that share the same unique key in order to compute a combined result. Since they work on independent inputs, Map and Reduce can be automatically processed in parallel, on different data partitions using many cluster nodes.

Figure 18.8 gives an overview of MapReduce execution in a cluster. There is one master node (not shown in the figure) in the cluster that assigns Map and Reduce tasks to cluster nodes, i.e., Map and Reduce nodes. The input data set is first automatically split into a number of partitions, each being processed by a different Map node that applies the Map operation to each input record to compute intermediate (key,value) pairs. The intermediate result is divided into n partitions, using a partitioning function applied to the key (e.g., hash(key) mod n). Map nodes periodically write to disk their intermediate data into n regions by applying the partitioning function and indicate the region locations to the master. Reduce nodes are assigned by the master to work on one or more partitions. Each Reduce node first reads the partitions from the corresponding regions on the Map nodes, disks, and groups the values by intermediate key, using sorting. Then, for each unique key and group of values, it calls the user Reduce operation to compute a final result that is written in the output data set.

As in the original description of MapReduce [Dean and Ghemawat, 2004], the favorite examples deal with sets of documents, e.g., counting the occurrences of each word in each document, or matching a given pattern in each document. However,

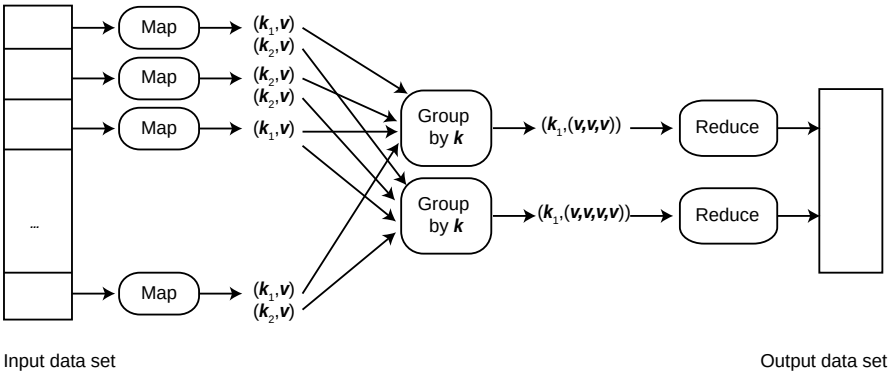


Fig. 18.8 Overview of MapReduce Execution

MapReduce can also be used to process relational data, as in the following example of a Group By select query on a single relation.

Example 18.3. Let us consider relation EMP(ENAME, TITLE, CITY) and the following SQL query that returns for each city, the number of employees whose name is "Smith".

```
SELECT   CITY, COUNT(*)
FROM     EMP
WHERE    ENAME LIKE "%Smith"
GROUP BY CITY
```

Processing this query with MapReduce can be done with the following Map and Reduce functions (which we give in pseudo code).

```
Map (Input (TID,emp), Output: (CITY,1))
   if emp.ENAME like "%Smith" return (CITY,1)
Reduce (Input (CITY,list(1)), Output: (CITY,SUM(list(1)))
   return (CITY,SUM(1*))
```

Map is applied in parallel to every tuple in EMP. It takes one pair (TID,emp), where the key is the EMP tuple identifier (TID) and the value the EMP tuple, and, if applicable, returns one pair (CITY,1). Note that the parsing of the tuple format to extract attributes needs to be done by the Map function. Then all (CITY,1) pairs with the same CITY are grouped together and a pair (CITY,list(1)) is created for each CITY. Reduce is then applied in parallel to compute the count for each CITY and produce the result of the query. ♦

Fault-tolerance is important as there may be many nodes executing Map and Reduce operations. Input and output data are stored in GFS that already provides high fault-tolerance. Furthermore, all intermediate data are written to disk that helps checkpointing Map operations, and thus provides tolerance to soft failures. However, if one Map node or Reduce node fails during execution (hard failure), the task can

be scheduled by the master onto other nodes. It may also be necessary to re-execute completed Map tasks, since the input data on the failed node disk is inaccessible. Overall, fault-tolerance is fine-grained and well suited for large jobs.

MapReduce has been extensively used both within Google and outside, with the Hadoop open source implementation, for many various applications including text processing, machine learning, and graph processing on very large data sets. The often cited advantages of MapReduce are its ability to express various (even complicated) Map and Reduce functions, and its extreme scalability and fault-tolerance. However, the comparison of MapReduce with parallel DBMSs in terms of performance has been the subject of debate between their respective proponents [Stonebraker et al., 2010; Dean and Ghemawat, 2010]. A performance comparison of Hadoop MapReduce and two parallel DBMSs – one row-store and one column-store DBMS – using a benchmark of three queries (a grep query, an aggregation query with a group by clause on a web log, and a complex join of two tables with aggregation and filtering) shows that, once the data has been loaded, the DBMSs are significantly faster, but loading data is very time consuming for the DBMSs [Pavlo et al., 2009]. The study also suggests that MapReduce is less efficient than DBMSs, because it performs repetitive format parsing and does not exploit pipelining and indices. It has been argued that a differentiation needs to be made between the MapReduce model and its implementations, which could be well improved, e.g., by exploiting indices [Dean and Ghemawat, 2010]. Another observation is that MapReduce and parallel DBMSs are complementary as MapReduce could be used to extract-transform-load data in a DBMS for more complex OLAP [Stonebraker et al., 2010].

18.3 Conclusion

In this chapter, we discussed two topics that are currently receiving considerable attention – data stream management, and cloud data management. Both of these have the potential to make considerable impact on distributed data management, but they are still not fully matured and require more research.

Data stream management addresses the requirements of a class of applications that produce data continuously. These systems require a shift in emphasis from traditional DBMSs in that they deal with data that is transient and queries that are (generally) persistent. Thus, they require new solutions and approaches. We discussed the main tenets of data stream management systems (DSMSs) in this chapter. The main challenge in data stream management is that data are produced continually, so it is not possible to store them for processing, as is typically done in traditional DBMSs. This requires unblocking operations, and online algorithms that sometimes have to deal with high data rates. The abstract models, language issues, and windowed query processing of streams are relatively well understood. However, there are a number of interesting research directions including the following:

- **Scaling with data rates.** Some data streams are relatively slow, while others have very high data rates. It is not clear if the strategies that have been developed

for processing queries work on the wide range of stream rates. It is probably the case that special processing techniques need to be developed for different classes of streams based on their data rates.

- **Distributed stream processing.** Although there has been some amount of work in considering processing streams in a distributed fashion, most of the existing works consider a single processing site. Distribution, as is usually the case, poses new challenges but also new opportunities that are worth exploring.
- **Stream data warehouses.** Stream data warehouses combine the challenges of standard data warehouses and data streams. This is an area that has recently started to receive attention (e.g., [Golab et al., 2009; Polyzotis et al., 2008]), but there are still many problems that require attention, including update scheduling strategies for optimizing various objectives, and monitoring data consistency and quality as new data arrive [Golab and Özsu, 2010].
- **Uncertain data streams.** In many applications that generate streaming data, there may be uncertainty in the data values. For example, sensors may be faulty and generate data that are not accurate, certain observations may be uncertain, etc. The processing of queries over uncertain data streams poses significant challenges that are still open.

One of the main challenges of cloud data management is to provide ease of programming, consistency, scalability and elasticity at the same time, over cloud data. Current solutions have been quite successful but developed with specific, relatively simple applications in mind. In particular, they have sacrificed consistency and ease of programming for the sake of scalability. This has resulted in a pervasive approach relying on data partitioning and forcing applications to access data partitions individually, with a loss of consistency guarantees across data partitions. As the need to support tighter consistency requirements, e.g., for updating multiple tuples in one or more tables, increases, cloud application developers will be faced with a very difficult problem: providing isolation and atomicity across data partitions through careful engineering. We believe that new solutions are needed that capitalize on the principles of distributed and parallel database systems to raise the level of consistency and abstraction, while retaining the scalability and simplicity advantages of current solutions. Parallel database management techniques such as pipelining, indices and optimization should also be useful to improve the performance of MapReduce-like systems and support more complex data analysis applications. In the context of large-scale shared-nothing clusters, where node failures become the norm rather than the exception, another important problem remains to deal with the trade-off between query performance and fault-tolerance. P2P techniques that do not require centralized query execution control by a master node could also be useful there. Some promising research directions for cloud data management include the following:

- **Declarative programming languages.** Programming large-scale, distributed data management software such as MapReduce remains very hard. One promising solution proposed in the BOOM project [Alvaro et al., 2010] is to adopt a data centric declarative programming language, based on the Overlog data

language, in order to improve ease of development and program correctness without sacrificing performance.

- **Autonomic data management.** Self-management of the data by the cloud will be critical to support large numbers of users with no database expertise. Modern database systems already provide good self-administration, self-tuning and self-repairing capabilities which ease application deployment and evolution. However, extending these capabilities to the scale of a cloud is hard. In particular, one problem is the automatic management of replication (definition, allocation, refreshment) to deal with load variations [Doherty and Hurley, 2007].
- **Data security and privacy.** Data security and access control in a cloud typically rely on user authentication and secured communication protocols to exchange encrypted data. However, the semi-open nature of a cloud makes security and privacy a major challenge since users may not trust the providers servers. Thus, the ability to perform relational-like operators directly on encrypted data at the cloud is important [Abadi, 2009]. In some applications, it is important that data privacy be preserved, using high-level mechanisms such as those of Hyppocratic databases [Agrawal et al., 2002].
- **Green data management.** One major problem for large-scale clouds is the energy cost. Harizopoulos et al. [2009] argue that data management techniques will be key in optimizing for energy efficiency. However, current data management techniques for the cloud have focused on scalability and performance, and must be significantly revisited to account for energy costs in query optimization, data structures and algorithms.

Finally there are problems in the intersection of data stream processing and cloud computing. Given the steady increase in data stream volumes, the need to process massive data flows in a scalable way is becoming important. Thus, the potential scalability advantage of a cloud can be exploited for data stream management as in Streamcloud [Gulisano et al., 2010]. This requires new strategies to parallelize continuous queries. And deadling with various trade-offs.

18.4 Bibliographic Notes

Data streams have received a lot of attention in recent years, so the literature on the topic is extensive. Good early overviews are given in [Babcock et al., 2002; Golab and Özsu, 2003a]. A more recent edited volume [Aggarwal, 2007] includes a number of articles on various aspects of these systems. An volume [Golab and Özsu, 2010] gives a full treatment of many of the issues that are discussed here. Mining data streams is reviewed in [Gaber et al., 2005] and issues in mining data streams with underlying distribution changes is discussed in [Hulten et al., 2001].

Our discussion of data stream systems follows [Golab and Özsu, 2003a], Chapter 2 of [Golab, 2006] and [Golab and Özsu, 2010]. The discussion on mining data streams borrows from Chapter 2 of [Tao, 2010].

Cloud computing has recently gained a lot of attention from the professional press as a new platform for enterprise and personal computing (see [Cusumano, 2010] for a good discussion of the trend). However, the research literature on cloud computing in general, and cloud data management in particular, is rather small, but as the number of international conferences and workshops grow, this should change quickly to become a major research domain. Our cloud taxonomy in Section 18.2.1 is based on our compilation of many professional articles and white papers. The discussion on grid computing in Section 18.2.2 is based on [Atkinson et al., 2005; Pacitti et al., 2007b]. The section on data management in the cloud (Section 18.2.4) has been inspired by several keynotes on the topic, e.g., [Ramakrishnan, 2009]. The technical details can be found in the research papers on GFS [Ghemawat et al., 2003], Bigtable [Chang et al., 2008], PNUTS [Cooper et al., 2008] and MapReduce [Dean and Ghemawat, 2004]. The discussion of MapReduce versus parallel DBMS can be found in [Stonebraker et al., 2010; Dean and Ghemawat, 2010].

References

Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: A new model and architecture for data stream management. *VLDB J.*, 12(2):120–139. 727, 730, 734, 736, 738

Abadi, D. J. (2009). Data management in the cloud: Limitations and opportunities. *Q. Bull. IEEE TC on Data Eng.*, 32(1):3–12. 746, 747, 762

Abadi, D. J., Madden, S., and Hachem, N. (2008). Column-stores vs. row-stores: how different are they really? In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 967–980. 753

Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer. 553, 607

Abbadi, A. E., Skeen, D., and Cristian, F. (1985). An efficient, fault–tolerant protocol for replicated data management. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 215–229. 488

Aberer, K. (2001). P-grid: A self-organizing access structure for p2p information systems. In *Proc. Int. Conf. on Cooperative Information Systems*, pages 179–194. 622

Aberer, K. (2003). Guest editor's introduction. *ACM SIGMOD Rec.*, 32(3):21–22. 653

Aberer, K., Cudré-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., and Schmidt, R. (2003a). P-grid: a self-organizing structured p2p system. *ACM SIGMOD Rec.*, 32(3):29–33. 622, 651, 654

Aberer, K., Cudré-Mauroux, P., and Hauswirth, M. (2003b). Start making sense: The chatty web approach for global semantic agreements. *J. Web Semantics*, 1(1):89–114. 625

Abiteboul, S. and Beeri, C. (1995). The power of languages for the manipulation of complex values. *VLDB J.*, 4(4):727–794. 553

Abiteboul, S., Benjelloun, O., Manolescu, I., Milo, T., and Weber, R. (2002). Active XML: Peer-to-peer data and web services integration. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 1087–1090. 625

Abiteboul, S., Benjelloun, O., and Milo, T. (2008a). The active XML project: an overview. *VLDB J.*, 17(5):1019–1040. 703

DOI 10.1007/978-1-4419-8834-8, © Springer Science+Business Media, LLC 2011

Abiteboul, S., Buneman, P., and Suciu, D. (1999). *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann. 719

Abiteboul, S. and dos Santos, C. S. (1995). IQL(2): A model with ubiquitous objects. In *Proc. 5th Int. Workshop on Database Programming Languages*, page 10. 607

Abiteboul, S. and Kanellakis, P. C. (1998a). Object identity as a query language primitive. *J. ACM*, 45(5):798–842. 553

Abiteboul, S. and Kanellakis, P. C. (1998b). Object identity as a query language primitive. *J. ACM*, 45(5):798–842. 607

Abiteboul, S., Manolescu, I., Polyzotis, N., Preda, N., and Sun, C. (2008b). XML processing in DHT networks. In *Proc. 24th Int. Conf. on Data Engineering*, pages 606–615. 625

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. (1997). The Lorel query language for semistructured data. *Int. J. Digit. Libr.*, 1(1):68–88. 673

Aboulnaga, A., Alameldeen, A. R., and Naughton, J. F. (2001). Estimating the selectivity of XML path expressions for internet scale applications. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 591–600. 701

Abramson, N. (1973). The ALOHA system. In Abramson, N. and Kuo, F. F., editors, *Computer Communication Networks*. Prentice-Hall. 64

Adali, S., Candan, K. S., Papakonstantinou, Y., and Subrahmanian, V. S. (1996a). Query caching and optimization in distributed mediator systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 137–148. 160

Adali, S., Candan, K. S., Papakonstantinou, Y., and Subrahmanian, V. S. (1996b). Query caching and optimization in distributed mediator systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 137–148. 309

Adamic, L. and Huberman, B. (2000). The nature of markets in the world wide web. *Quart. J. Electron. Comm.*, 1:5–12. 734, 739

Adiba, M. (1981). Derived relations: A unified mechanism for views, snapshots and distributed data. In *Proc. 7th Int. Conf. on Very Data Bases*, pages 293–305. 176, 177, 201

Adiba, M. and Lindsay, B. (1980). Database snapshots. In *Proc. 6th Int. Conf. on Very Data Bases*, pages 86–91. 176, 201

Adler, M. and Mitzenmacher, M. (2001). Towards compressing web graphs. In *Proc. Data Compression Conf.*, pages 203–212. 660, 719

Adya, A., Gruber, R., Liskov, B., and Maheshwari, U. (1995). Efficient optimistic concurrency control using loosely synchronized clocks. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 23–34. 574

Aggarwal, C. (2003). A framework for diagnosing changes in evolving data streams. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 575–586. 743

Aggarwal, C. (2005). On change diagnosis in evolving data streams. *IEEE Trans. Knowl. and Data Eng.*, 17(5). 743

Aggarwal, C., Han, J., Wang, J., and Yu, P. S. (2003). A framework for clustering evolving data streams. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 81–92. 743

Aggarwal, C., Han, J., Wang, J., and Yu, P. S. (2004). A framework for projected clustering of high dimensional data streams. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 852–863. 726, 743

Aggarwal, C. C., editor (2007). *Data Streams: Models and Algorithms*. Springer. 762

Agichtein, E., Lawrence, S., and Gravano, L. (2004). Learning to find answers to questions on the web. *ACM Trans. Internet Tech.*, 4(3):129—162. 681

Agrawal, D., Bruno, J. L., El-Abbadi, A., and Krishnasawamy, V. (1994). Relative serializability: An approach for relaxing the atomicity of transactions. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 139–149. 395

Agrawal, D. and El-Abbadi, A. (1990). Locks with constrained sharing. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 85–93. 371, 372

Agrawal, D. and El-Abbadi, A. (1994). A nonrestrictive concurrency control protocol for object-oriented databases. *Distrib. Parall. Databases*, 2(1):7–31. 600

Agrawal, R., Carey, M., and Livney, M. (1987). Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654. 401

Agrawal, R. and DeWitt, D. J. (1985). Integrated concurrency control and recovery mechanisms. *ACM Trans. Database Syst.*, 10(4):529–564. 420

Agrawal, R., Evfimievski, A. V., and Srikant, R. (2003). Information sharing across private databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 86–97. 187

Agrawal, R., Ghosh, S. P., Imielinski, T., Iyer, B. R., and Swami, A. N. (1992). An interval classifier for database mining applications. In *Proc. 18th Int. Conf. on Very Large Data Bases*, pages 560–573. 743

Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. (2002). Hippocratic databases. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 143–154. 762

Akal, F., Böhm, K., and Schek, H.-J. (2002). Olap query evaluation in a database cluster: A performance study on intra-query parallelism. In *Proc. 6th East European Conf. Advances in Databases and Information Systems*, pages 218–231. 542, 543, 548

Akal, F., Türker, C., Schek, H.-J., Breitbart, Y., Grabs, T., and Veen, L. (2005). Fine-grained replication and scheduling with freshness and correctness guarantees. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 565–576. 493

Akbarinia, R. and Martins, V. (2007). Data management in the appa system. *J. Grid Comp.*, 5(3):303–317. 626

Akbarinia, R., Martins, V., Pacitti, E., and Valduriez, P. (2006a). Design and implementation of atlas p2p architecture. In Baldoni, R., Cortese, G., and Davide, F., editors, *Global Data Management*, pages 98–123. IOS Press. 626, 636

Akbarinia, R., Pacitti, E., and Valduriez, P. (2006b). Reducing network traffic in unstructured p2p systems using top-k queries. *Distrib. Parall. Databases*, 19(2-3):67–86. 628, 637

Akbarinia, R., Pacitti, E., and Valduriez, P. (2007a). Best position algorithms for top-k queries. In *Proc. 33rd Int. Conf. on Very Large Data Bases*, pages 495–506. 634, 635, 654

Akbarinia, R., Pacitti, E., and Valduriez, P. (2007b). Data currency in replicated dhts. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 211–222. 648, 654

Akbarinia, R., Pacitti, E., and Valduriez, P. (2007c). Processing top-k queries in distributed hash tables. In *Proc. 13th Int. Euro-Par Conf.*, pages 489–502. 638, 654

Akbarinia, R., Pacitti, E., and Valduriez, P. (2007d). Query processing in P2P systems. Technical Report 6112, INRIA, Rennes, France. 654

Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N., and Srivastava, D. (2002). Structural joins: A primitive for efficient XML query pattern matching. In *Proc. 18th Int. Conf. on Data Engineering*, pages 141–152. 700

Alon, N., Matias, Y., and Szegedy, M. (1996). The space complexity of approximating the frequency moments. In *Proc. 28th Annual ACM Symp. on Theory of Computing*, pages 20–29. 733

Alsberg, P. A. and Day, J. D. (1976). A principle for resilient sharing of distributed resources. In *Proc. 2nd Int. Conf. on Software Engineering*, pages 562–570. 373

Altingövde, I. S. and Ulusoy, Ö. (2004). Exploiting interclass rules for focused crawling. *IEEE Intelligent Systems*, 19(6):66–73. 666

Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J. M., and Sears, R. (2010). Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proc. 5th ACM SIGOPS/EuroSys European Conf. on Computer Systems*, pages 223–236. 761

Amsaleg, L. (1995). *Conception et réalisation d'un glaneur de cellules adapté aux SGBDO client-serveur*. Ph.D. thesis, Université Paris 6 Pierre et Marie Curie, Paris, France. 581

Amsaleg, L., Franklin, M., and Gruber, O. (1995). Efficient incremental garbage collection for client-server object database systems. In *Proc. 21th Int. Conf. on Very Large Data Bases*, pages 42–53. 581

Amsaleg, L., Franklin, M. J., Tomasic, A., and Urhan, T. (1996a). Scrambling query plans to cope with unexpected delays. In *Proc. 4th Int. Conf. on Parallel and Distributed Information Systems*, pages 208–219. 320, 322, 331

Amsaleg, L., Franklin, M. J., Tomasic, A., and Urhan, T. (1996b). Scrambling query plans to cope with unexpected delays. In *Proc. 4th Int. Conf. on Parallel and Distributed Information Systems*, pages 208–219. 739

Anderson, T. and Lee, P. A. (1981). *Fault Tolerance: Principles and Practice*. Prentice-Hall. 455

Anderson, T. and Lee, P. A. (1985). Software fault tolerance terminology proposals. In Shrivastava [1985], pages 6–13. 406

Anderson, T. and Randell, B. (1979). *Computing Systems Reliability*. Cambridge University Press. 455

ANSI (1992). *Database Language SQL*, ansi x3.135-1992 edition. 348

ANSI/SPARC (1975). Interim report: ANSI/X3/SPARC study group on data base management systems. *ACM FDT Bull*, 7(2):1–140. 22

Antonioletti, M. et al. (2005). The design and implementation of grid database services in OGSA-DAI. *Concurrency — Practice & Experience*, 17(2-4):357–376. 750

Apers, P., van den Berg, C., Flokstra, J., Grefen, P., Kersten, M., and Wilschut, A. (1992). Prisma/db: a parallel main-memory relational dbms. *IEEE Trans. Knowl. and Data Eng.*, 4:541–554. 505, 548

Apers, P. M. G. (1981). Redundant allocation of relations in a communication network. In *Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 245–258. 125

Apers, P. M. G., Hevner, A. R., and Yao, S. B. (1983). Optimization algorithms for distributed queries. *IEEE Trans. Softw. Eng.*, 9(1):57–68. 212

Arasu, A., Babu, S., and Widom, J. (2006). The CQL continuous query language: Semantic foundations and query execution. *VLDB J.*, 15(2):121–142. 726, 727, 728, 732, 734, 735, 739

Arasu, A., Cho, J., Garcia-Molina, H., Paepcke, A., and Raghavan, S. (2001). Searching the web. *ACM Trans. Internet Tech.*, 1(1):2–43. 663, 667, 719

Arasu, A. and Widom, J. (2004a). A denotational semantics for continuous queries over streams and relations. *ACM SIGMOD Rec.*, 33(3):6–11. 728

Arasu, A. and Widom, J. (2004b). Resource sharing in continuous sliding-window aggregates. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 336–347. 736, 737

Arocena, G. and Mendelzon, A. (1998). Weboql: Restructuring documents, databases and webs. In *Proc. 14th Int. Conf. on Data Engineering*, pages 24–33. 676

Arpaci-Dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. (1999). Cluster i/o with river: making the fast case common. In *Proc. Workshop on I/O in Parallel and Distributed Systems*, pages 10–22. 326

Aspnes, J. and Shah, G. (2003). Skip graphs. In *Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 384–393. 622

Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V. (1976). System r: A relational database management system. *ACM Trans. Database Syst.*, 1(2):97–137. 190, 261, 419

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1989). The object-oriented database system manifesto. In *Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases*, pages 40–57. 553

Atkinson, M. P. et al. (2005). Web service grids: an evolutionary approach. *Concurrency and Computation — Practice & Experience*, 17(2-4):377–389. 748, 763

Avizienis, A., Kopetz, H., and (eds.), J. C. L. (1987). *The Evolution of Fault-Tolerant Computing*. Springer. 455

Avnur, R. and Hellerstein, J. M. (2000). Eddies: Continuously adaptive query processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 261–272. 321, 331

Ayad, A. and Naughton, J. (2004). Static optimization of conjunctive queries with sliding windows over unbounded streaming information sources. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 419–430. 740

Ayad, A., Naughton, J., Wright, S., and Srivastava, U. (2006). Approximate streaming window joins under CPU limitations. In *Proc. 22nd Int. Conf. on Data Engineering*, page 142. 740

Babaoglu, Ö. (1987). On the reliability of consensus-based fault-tolerant distributed computing systems. *ACM Trans. Comp. Syst.*, 5(3):394–416. 456

Babb, E. (1979). Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst.*, 4(1):1–29. 499

Babcock, B., Babu, S., Datar, M., Motwani, R., and Thomas, D. (2004). Operator scheduling in data stream systems. *VLDB J.*, 13(4):333–353. 735

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 1–16. 740, 743, 762

Babcock, B., Datar, M., Motwani, R., and O'Callaghan, L. (2003). Maintaining variance and k-medians over data stream windows. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 234–243. 737

Babu, S. and Bizarro, P. (2005). Adaptive query processing in the looking glass. In *Proc. 2nd Biennial Conf. on Innovative Data Systems Research*, pages 238–249. 739

Babu, S., Motwani, R., Munagala, K., Nishizawa, I., and Widom, J. (2004a). Adaptive ordering of pipelined stream filters. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 407–418. 739

Babu, S., Munagala, K., Widom, J., and Motwani, R. (2005). Adaptive caching for continuous queries. In *Proc. 21st Int. Conf. on Data Engineering*, pages 118–129. 739

Babu, S., Srivastava, U., and Widom, J. (2004b). Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580. 732

Babu, S. and Widom, J. (2004). StreaMon: an adaptive engine for stream query processing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 931–932. 739

Badrinath, B. R. and Ramamritham, K. (1987). Semantics-based concurrency control: Beyond commutativity. In *Proc. 3th Int. Conf. on Data Engineering*, pages 04–311. 594, 596, 602

Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison Wesley, New York, USA. 669

Balke, W.-T., Nejdl, W., Siberski, W., and Thaden, U. (2005). Progressive distributed top-k retrieval in peer-to-peer networks. In *Proc. 21st Int. Conf. on Data Engineering*, pages 174–185. 639

Ball, M. O. and Hardie, F. (1967). Effects and detection of intermittent failures in digital systems. Technical Report Internal Report 67-825-2137, IBM. Cited in [Siewiorek and Swarz, 1982]. 410

Balter, R., Berard, P., and Decitre, P. (1982). Why control of concurrency level in distributed systems is more important than deadlock management. In *Proc. ACM SIGACT-SIGOPS 1st Symp. on the Principles of Distributed Computing*, pages 183–193. 361

Bancilhon, F. and Spyratos, N. (1981). Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575. 175, 201

Barbara, D., Garcia-Molina, H., and Spauster, A. (1986). Policies for dynamic vote reassignment. In *Proc. 6th Int. Conf. on Distributed Computing Systems*, pages 37–44. 456, 493

Barbara, D., Molina, H. G., and Spauster, A. (1989). Increasing availability under mutual exclusion constraints with dynamic voting reassignment. *ACM Trans. Comp. Syst.*, 7(4):394–426. 456, 493

Bartlett, J. (1978). A nonstop operating system. In *Proc. 11th Hawaii Int. Conf. on System Sciences*, pages 103–117. 456

Bartlett, J. (1981). A nonstop kernel. In *Proc. 8th ACM Symp. on Operating System Principles*, pages 22–29. 456

Barton, C., Charles, P., Goyal, D., Raghavachari, M., Fontoura, M., and Josifovski, V. (2003). Streaming XPath processing with forward and backward axes. In *Proc. 19th Int. Conf. on Data Engineering*, pages 455–466. 700

Batini, C. and Lenzirini, M. (1984). A methodology for data schema integration in entity-relationship model. *IEEE Trans. Softw. Eng.*, SE-10(6):650–654. 147

Batini, C., Lenzirini, M., and Navathe, S. B. (1986). A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364. 140, 147, 160

Bayer, R. and McCreight, E. (1972). Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189. 510

Beeri, C. (1990). A formal approach to object-oriented databases. *Data & Knowledge Eng*, 5:353–382. 557

Beeri, C., Bernstein, P. A., and Goodman, N. (1989). A model for concurrency in nested transaction systems. *J. ACM*, 36(2):230–269. 401

Beeri, C., Schek, H.-J., and Weikum, G. (1988). Multi-level transaction management, theoretical art or practical need? In *Advances in Database Technology, Proc. 1st Int. Conf. on Extending Database Technology*, pages 134–154. 397

Bell, D. and Grimson, J. (1992). *Distributed Database Systems*. Addison Wesley. Reading. 38

Bell, D. and Lapuda, L. (1976). Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997 Rev.1, MITRE Corp, Bedford, MA. 183, 201

Bellatreche, L., Karlapalem, K., and Li, Q. (1998). Complex methods and class allocation in distributed object oriented database systems. Technical Report HKUST98-yy, Department of Computer Science, Hong Kong University of Science and Technologyty of Science and Technology. 565

Bellatreche, L., Karlapalem, K., and Li, Q. (2000a). Algorithms and support for horizontal class partitioning in object-oriented databases. *Distrib. Parall. Databases*, 8(2):155 – 179. 607

Bellatreche, L., Karlapalem, K., and Li, Q. (2000b). A framework for class partitioning in object oriented databases. *Distrib. Parall. Databases*, 8(2):333 – 366. 607

Benzaken, V. and Delobel, C. (1990). Enhancing performance in a persistent object store: Clustering strategies in o_2. In *Implementing Persistent Object Bases: Principles and Practice. Proc. 4th Int. Workshop on Persistent Object Systems*, pages 403–412. 579

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ansi sql isolation levels. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–10. 348, 349, 367

Bergamaschi, S., Castano, S., Vincini, M., and Beneventano, D. (2001). Semantic integration of heterogeneous information sources. *Data & Knowl. Eng.*, 36:215–249. 134, 160

Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., and Siméon, J., editors. XML Path language (XPath) 2.0 (2007). Available from: `http://www.w3.org/TR/xpath20/` [Last retrieved: December 2009]. 690, 694

Bergman, M. K. (2001). The deep web: Surfacing hidden value. *J. Electronic Publishing*, 7(1). 657

Bergsten, B., Couprie, M., and Valduriez, P. (1991). Prototyping dbs3, a shared-memory parallel database system. In *Proc. Int. Conf. on Parallel and Distributed Information Systems*, pages 226–234. 501, 503, 528, 548

Bergsten, B., Couprie, M., and Valduriez, P. (1993). Overview of parallel architectures for databases. *The Comp. J.*, 36(8):734–739. 547

Berlin, J. and Motro, A. (2001). Autoplex: Automated discovery of content for virtual databases. In *Proc. Int. Conf. on Cooperative Information Systems*, pages 108–122. 145

Bernstein, P. and Blaustein, B. (1982). Fast methods for testing quantified relational calculus assertions. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 39–50. 192, 199, 202

Bernstein, P., Blaustein, B., and Clarke, E. M. (1980a). Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proc. 6th Int. Conf. on Very Data Bases*, pages 126–136. 192, 202

Bernstein, P. and Melnik, S. (2007). Model management: 2.0: Manipulating richer mappings. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12. 135, 159, 160

Bernstein, P., Shipman, P., and Rothnie, J. B. (1980b). Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):18–51. 383, 395

Bernstein, P. A. and Chiu, D. M. (1981). Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40. 269, 292

Bernstein, P. A., Fekete, A., Guo, H., Ramakrishnan, R., and Tamma, P. (2006). Relexed concurrency serializability for middle-tier caching and replication. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 599–610. 462, 464, 480

Bernstein, P. A., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., and Zaihrayeu, I. (2002). Data management for peer-to-peer computing : A vision. In *Proc. 5th Int. Workshop on the World Wide Web and Databases*, pages 89–94. 625, 653

Bernstein, P. A. and Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–222. 39, 367, 369, 401

Bernstein, P. A. and Goodman, N. (1984). An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*, 9(4):596–615. 486

Bernstein, P. A., Goodman, N., Wong, E., Reeve, C. L., and Jr, J. B. R. (1981). Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 6(4):602–625. 215, 281, 283, 293

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison Wesley. 39, 341, 385, 391, 401, 413, 421, 423, 424, 425, 429, 453, 486, 596

Bernstein, P. A. and Newcomer, E. (1997). *Principles of Transaction Processing for the Systems Professional*. Morgan Kaufmann. 358

Berthold, H., Schmidt, S., Lehner, W., and Hamann, C.-J. (2005). Integrated resource management for data stream systems. In *Proc. 2005 ACM Symp. on Applied Computing*, pages 555–562. 738

Bertino, E., Chin, O. B., Sacks-Davis, R., Tan, K.-L., Zobel, J., Shidlovsky, B., and Andronico, D. (1997). *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers. 607

Bertino, E. and Kim, W. (1989). Indexing techniques for queries on nested objects. *IEEE Trans. Knowl. and Data Eng.*, 1(2):196–214. 588, 589, 590

Bertino, E. and Martino, L. (1993). *Object-Oriented Database Systems*. Addison Wesley. 607

Bevan, D. I. (1987). Distributed garbage collection using reference counting. In de Bakker, J., Nijman, L., and Treleaven, P., editors, *Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, pages 117–187. Springer. 581

Bhar, S. and Barker, K. (1995). Static allocation in distributed objectbase systems: A graphical approach. In *Proc. 6th Int. Conf. on Information Systems and Data Management*, pages 92–114. 565

Bharat, K. and Broder, A. (1998). A technique for measuring the relative size and overlap of public web search engines. *Comp. Networks and ISDN Syst.*, 30:379 – 388. (Proc. 7th Int. World Wide Web Conf.). 657

Bhargava, B., editor (1987). *Concurrency Control and Reliability in Distributed Systems*. Van Nostrand Reinhold. 358

Bhargava, B. and Lian, S.-R. (1988). Independent checkpointing and concurrent rollback for recovery in distributed systems: An optimistic approach. In *Proc. 7th Symp. on Reliable Distributed Systems*, pages 3–12. 456

Bhide, A. (1988). An analysis of three transaction processing architectures. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 339–350. 401

Bhide, A. and Stonebraker, M. (1988). A performance comparison of two architectures for fast transaction processing. In *Proc. 4th Int. Conf. on Data Engineering*, pages 536–545. 547

Bhowmick, S. S., Madria, S. K., and Ng, W. K. (2004). *Web Data Management*. Springer. 719

Biliris, A. and Panagos, E. (1995). A high performance configurable storage manager. In *Proc. 11th Int. Conf. on Data Engineering*, pages 35–43. 571

Biscondi, N., Brunie, L., Flory, A., and Kosch, H. (1996). Encapsulation of intra-operation parallelism in a parallel match operator. In *Proc. ACPC Conf.*, volume 1127 of *Lecture Notes in Computer Science*, pages 124–135. 528

Bitton, D., Boral, H., DeWitt, D. J., and Wilkinson, W. K. (1983). Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst.*, 8(3):324–353. 515

Blakeley, J., McKenna, W., and Graefe, G. (1993). Experiences building the open oodb query optimizer. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 287–296. 584, 586, 587, 588

Blakeley, J. A., Larson, P.-A., and Tompa, F. W. (1986). Efficiently updating materialized views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 61–71. 177

Blasgen, M., Gray, J., Mitoma, M., and Price, T. (1979). The convoy phenomenon. *Operating Systems Rev.*, 13(2):20–25. 526

Blaustein, B. (1981). *Enforcing Database Assertions: Techniques and Applications*. Ph.D. thesis, Harvard University, Cambridge, Mass. 192, 202

Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J., editors. XQuery 1.0: An XML query language (2007). Available from: `http://www.w3.org/TR/xquery` [Last retrieved: December 2009]. 690, 694, 696

Bonato, A. (2008). *A Course on the Web Graph*. American Mathematical Society. 658, 719

Boncz, P. A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., and Teubner, J. (2006). MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 479–490. 699, 703

Bonnet, P., Gehrke, J., and Seshadri, P. (2001). Towards sensor database systems. In *Proc. 2nd Int. Conf. on Mobile Data Management*, pages 3–14. 726, 730

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D., editors. Web services architecture (2004). Available from: `http://www.w3.org/TR/ws-arch/` [Last retrieved: December 2009]. 690

Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., and Valduriez, P. (1990). Prototyping bubba, a highly parallel database system. *IEEE Trans. Knowl. and Data Eng.*, 2(1):4–24. 505

Boral, H. and DeWitt, D. (1983). Database machines: An idea whose time has passed? a critique of the future of database machines. In *Proc. 3rd Int. Workshop on Database Machines*, pages 166–187. 498

Borg, A., Baumbach, J., and Glazer, S. (1983). A message system supporting fault tolerance. In *Proc. 9th ACM Symp. on Operating System Principles*, pages 90–99, Bretton Woods, N.H. 456

Borr, A. (1984). Robustness to crash in a distributed database: A non shared-memory multiprocessor approach. In *Proc. 10th Int. Conf. on Very Large Data Bases*, pages 445–453. 456

Borr, A. (1988). High performance sql through low-level system integration. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 342–349. 377

Bouganim, L., Dageville, B., and Florescu, D. (1996a). Skew handling in the dbs3 parallel database system. In *Proc. International Conference on ACPC*. 528

Bouganim, L., Dageville, B., and Valduriez, P. (1996b). Adaptative parallel query execution in dbs3. In *Advances in Database Technology, Proc. 5th Int. Conf. on Extending Database Technology*, pages 481–484. Springer. 528, 548

Bouganim, L., Florescu, D., and Valduriez, P. (1996c). Dynamic load balancing in hierarchical parallel database systems. In *Proc. 22th Int. Conf. on Very Large Data Bases*, pages 436–447. 530, 534, 548

Bouganim, L., Florescu, D., and Valduriez, P. (1999). Multi-join query execution with skew in numa multiprocessors. *Distrib. Parall. Databases*, 7(1). in press. 506, 548

Brantner, M., Helmer, S., Kanne, C.-C., and Moerkotte, G. (2005). Full-fledged algebraic XPath processing in natix. In *Proc. 21st Int. Conf. on Data Engineering*, pages 705–716. 698, 700, 703

Bratbergsengen, K. (1984). Hashing methods and relational algebra operations. In *Proc. 10th Int. Conf. on Very Large Data Bases*, pages 323–333. 211, 515

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F., editors. Extensible markup language (XML) 1.0 (Fifth edition) (2008). Available from: `http://www.w3.org/TR/2008/REC-xml-20081126/` [Last retrieved: December 2009]. 689

Breitbart, Y. and Korth, H. F. (1997). Replication and consistency: Being lazy helps sometimes. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 173–184. 476

Breitbart, Y., Olson, P. L., and Thompson, G. R. (1986). Database integration in a distributed heterogeneous database system. In *Proc. 2nd Int. Conf. on Data Engineering*, pages 301–310. 160

Bright, M. W., Hurson, A. R., and Pakzad, S. H. (1994). Automated resolution of semantic heterogeneity in multidatabases. *ACM Trans. Database Syst.*, 19(2):212–253. 160

Brill, D., Templeton, M., and Yu, C. (1984). Distributed query processing strategies in mermaid: A front-end to data management systems. In *Proc. 1st Int. Conf. on Data Engineering*, pages 211–218. 331

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Comp. Netw.*, 30(1-7):107 – 117. 658, 667

Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J. (2000). Graph structure in the web. *Comp. Netw.*, 33:309–320. 659

Bruno, N. and Chaudhuri, S. (2002). Exploiting statistics on query expressions for optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 263–274. 256

Bruno, N., Koudas, N., and Srivastava, D. (2002). Holistic twig joins: Optimal XML pattern matching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 310–322. 700, 701

Bucci, G. and Golinelli, S. (1977). A distributed strategy for resource allocation in information networks. In *Proc. Int. Computing Symp*, pages 345–356. 125

Buchmann, A., Özsu, M., Hornick, M., Georgakopoulos, D., and Manola, F. A. (1982). A transaction model for active distributed object systems. In [Elmagarmid, 1982]. 354, 355, 359, 593, 594

Buneman, P., Cong, G., Fan, W., and Kementsietsidis, A. (2006). Using partial evaluation in distributed query evaluation. In *Proc. 32nd Int. Conf. on Very Large Data Bases*, pages 211–222. 711

Buneman, P., Davidson, S., Hillebrand, G. G., and Suciu, D. (1996). A query language and optimization techniques for unstructured data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 505–516. 671

Butler, M. (1987). Storage reclamation in object oriented database systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 410–425. 581

Calì, A. and Calvanese, D. (2002). Optimized querying of integrated data over the web. In *Engineering Information Systems in the Internet Context*, pages 285–301. 303

Callan, J. P. and Connell, M. E. (2001). Query-based sampling of text databases. *ACM Trans. Information Syst.*, 19(2):97–130. 688

Callan, J. P., Connell, M. E., and Du, A. (1999). Automatic discovery of language models for text databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 479–490. 688

Cammert, M., Krämer, J., Seeger, B., and S.Vaupel (2006). An approach to adaptive memory management in data stream systems. In *Proc. 22nd Int. Conf. on Data Engineering*, page 137. 735, 740

Canaday, R. H., Harrisson, R. D., Ivie, E. L., Rydery, J. L., and Wehr, L. A. (1974). A back-end computer for data base management. *Commun. ACM*, 17(10):575–582. 30, 547

Cao, P. and Wang, Z. (2004). Query processing issues in image (multimedia) databases. In *ACM Symp. on Principles of Distributed Computing (PODC)*, pages 206–215. 631, 633

Carey, M., Franklin, M., and Zaharioudakis, M. (1997). Adaptive, fine-grained sharing in a client-server oodbms: A callback-based approach. *ACM Trans. Database Syst.*, 22(4):570–627. 572

Carey, M. and Lu, H. (1986). Load balancing in a locally distributed database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 108–119. 287, 288, 293

Carey, M. and Stonebraker, M. (1984). The performance of concurrency control algorithms for database management systems. In *Proc. 10th Int. Conf. on Very Large Data Bases*, pages 107–118. 401

Carey, M. J., DeWitt, D. J., Franklin, M. J., Hall, N. E., McAuliffe, M. L., Naughton, J. F., Schuh, D. T., Solomon, M. H., Tan, C. K., Tsatalos, O. G., White, S. J., and Zwilling, M. J. (1994). Shoring up persistent applications. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 383–394. 571

Carey, M. J., Franklin, M., Livny, M., and Shekita, E. (1991). Data caching tradeoffs in client-server dbms architectures. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 357–366. 573

Carey, M. J. and Livny, M. (1988). Distributed concurrency control performance: A study of algorithms, distribution and replication. In *Proc. 14th Int. Conf. on Very Large Data Bases*, pages 13–25. 400, 401

Carey, M. J. and Livny, M. (1991). Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, 16(4):703–746. 401

Carney, D., Cetintemel, U., Rasin, A., Zdonik, S., Cherniack, M., and Stonebraker, M. (2003). Operator scheduling in a data stream manager. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 838–849. 735

Cart, M. and Ferrie, J. (1990). Integrating concurrency control into an object-oriented database system. In *Advances in Database Technology, Proc. 2nd Int. Conf. on Extending Database Technology*, pages 363–377. Springer. 597

Casey, R. G. (1972). Allocation of copies of a file in an information network. In *Proc. Spring Joint Computer Conf*, pages 617–625. 115

Castano, S. and Antonellis, V. D. (1999). A schema analysis and reconciliation tool environment for heterogeneous databases. In *Proc. Int. Conf. on Database Eng. and Applications*, pages 53–62. 134

Castano, S., Fugini, M. G., Martella, G., and Samarati, P. (1995). *Database Security*. Addison Wesley. 180, 201

Castro, M., Adya, A., Liskov, B., and Myers, A. (1997). Hac: Hybrid adaptive caching for distributed storage systems. In *Proc. ACM Symp. on Operating System Principles*, pages 102–115. 570

Cattell, R. G., Barry, D. K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., and Velez, F. (2000). *The Object Database Standard: ODMG-3.0*. Morgan Kaufmann. 553, 582

Cattell, R. G. G. (1994). *Object Data Management*. Addison Wesley, 2 edition. 607

Cellary, W., Gelenbe, E., and Morzy, T. (1988). *Concurrency Control in Distributed Database Systems*. North-Holland. 358, 401

Ceri, S., Gottlob, G., and Pelagatti, G. (1986). Taxonomy and formal properties of distributed joins. *Inf. Syst.*, 11(1):25–40. 232, 234, 242

Ceri, S., Martella, G., and Pelagatti, G. (1982a). Optimal file allocation in a computer network: A solution method based on the knapsack problem. *Comp. Netw.*, 6:345–357. 121

Ceri, S. and Navathe, S. B. (1983). A methodology for the distribution design of databases. *Digest of Papers - COMPCON*, pages 426–431. 125

Ceri, S., Navathe, S. B., and Wiederhold, G. (1983). Distribution design of logical database schemes. *IEEE Trans. Softw. Eng.*, SE-9(4):487–503. 81, 82, 121

Ceri, S., Negri, M., and Pelagatti, G. (1982b). Horizontal data partitioning in database design. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 128–136. 84, 87

Ceri, S. and Owicki, S. (1982). On the use of optimistic methods for concurrency control in distributed databases. In *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–130. 385

Ceri, S. and Pelagatti, G. (1982). A solution method for the non-additive resource allocation problem in distributed system design. *Inf. Proc. Letters*, 15(4):174–178. 125

Ceri, S. and Pelagatti, G. (1983). Correctness of query execution strategies in distributed databases. *ACM Trans. Database Syst.*, 8(4):577–607. 38, 232, 242, 292

Ceri, S. and Pelagatti, G. (1984). *Distributed Databases: Principles and Systems*. McGraw-Hill. 84, 220

Ceri, S. and Pernici, B. (1985). Dataid–d: Methodology for distributed database design. In Albano, V. d. A. and di Leva, A., editors, *Computer-Aided Database Design*, pages 157–183. North-Holland. 121

Ceri, S., Pernici, B., and Wiederhold, G. (1987). Distributed database design methodologies. *Proc. IEEE*, 75(5):533–546. 38, 73, 125

Ceri, S. and Widom, J. (1993). Managing semantic heterogeneity with production rules and persistent queues. In *Proc. 19th Int. Conf. on Very Large Data Bases*, pages 108–119. 160

Chakrabarti, K., Keogh, E., Mehrotra, S., and Pazzani, M. (2002). Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.*, 27. 666, 743

Chakrabarti, S., Dom, B., and Indyk, P. (1998). Enhanced hypertext classification using hyperlinks. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 307 – 318. 658

Chamberlin, D., Gray, J., and Traiger, I. (1975). Views, authorization and locking in a relational database system. In *Proc. National Computer Conf*, pages 425–430. 172, 201

Chamberlin, D. D., Astrahan, M. M., King, W. F., Lorie, R. A., Mehl, J. W., Price, T. G., Schkolnick, M., Selinger, P. G., Slutz, D. R., Wade, B. W., and Yost, R. A. (1981). Support for repetitive transactions and ad hoc queries in System R. *ACM Trans. Database Syst.*, 6(1):70–94. 265

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. (2003). TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. 1st Biennial Conf. on Innovative Data Systems Research*, pages 269–280. 728, 736

Chandrasekaran, S. and Franklin, M. J. (2003). PSoup: a system for streaming queries over streaming data. *VLDB J.*, 12(2):140–156. 736, 741

Chandrasekaran, S. and Franklin, M. J. (2004). Remembrance of streams past: overload-sensitive management of archived streams. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 348–359. 738

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comp. Syst.*, 26(2). 755, 756, 763

Chang, S. K. and Cheng, W. H. (1980). A methodology for structured database decomposition. *IEEE Trans. Softw. Eng.*, SE-6(2):205–218. 123

Chang, S. K. and Liu, A. C. (1982). File allocation in a distributed database. *Int. J. Comput. Inf. Sci*, 11(5):325–340. 121, 123

Charikar, M., Chen, K., and Motwani, R. (1997). Incremental clustering and dynamic information retrieval. In *Proc. 29th Annual ACM Symp. on Theory of Computing*. 743

Charikar, M., O'Callaghan, L., and Panigrahy, R. (2003). Better streaming algorithms for clustering problems. In *Proc. 35th Annual ACM Symp. on Theory of Computing*. 743

Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 34–43. 292

Chaudhuri, S., Ganjam, K., Ganti, V., and Motwani, R. (2003). Robust and efficient fuzzy match for online data cleaning. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 313–324. 158

Chen, J., DeWitt, D., and Naughton, J. (2002). Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *Proc. 18th Int. Conf. on Data Engineering*, pages 345–357. 740

Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000). NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 379–390. 6, 740

Chen, P. P. S. (1976). The entity-relationship model: Towards a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36. 81, 136

Chen, S., Deng, Y., Attie, P., and Sun, W. (1996). Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs. In *Proc. IEEE Int. Conf. Dist. Comp. Sys*, pages 613–619. 401

Chen, W. and Warren, D. S. (1989). C-logic of complex objects. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 369–378. 607

Cheng, J. M. et al. (1984). Ibm database 2 performance : Design, implementation and tuning. *IBM Systems J.*, 23(2):189–210. 503

Chiu, D. M. and Ho, Y. C. (1980). A methodology for interpreting tree queries into optimal semi-join expressions. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 169–178. 271, 272, 292

Cho, J. and Garcia-Molina, H. (2000). The evolution of the web and implications for an incremental crawler. In *Proc. 26th Int. Conf. on Very Large Data Bases*. 666

Cho, J. and Garcia-Molina, H. (2002). Parallel crawlers. In *Proc. 11th Int. World Wide Web Conf.* 666