15.6.2.1 Algebraic Optimization

Search Space and Transformation Rules.

The transformation rules are very much dependent upon the specific object algebra, since they are defined individually for each object algebra and for their combinations. The general considerations for the definition of transformation rules and the manipulation of query expressions is quite similar to relational systems, with one particularly important difference. Relational query expressions are defined on flat relations, whereas object queries are defined on classes (or collections or sets of objects) that have subclass and composition relationships among them. It is, therefore, possible to use the semantics of these relationships in object query optimizers to achieve some additional transformations.

Consider, for example, three object algebra operators [Straube and Özsu, 1990a]: `union` (denoted $\cup$), `intersection` (denoted $\cap$) and parameterized `select` (denoted $P\sigma_F < Q_1 \ldots Q_k >$), where `union` and `intersection` have the usual set-theoretic semantics, and `select` selects objects from one set P using the sets of objects $Q_1 \ldots Q_k$ as parameters (in a sense, a generalized form of semijoin). The results of these operators are sets of objects as well. It is, of course, possible to specify the usual set-theoretic, syntactic rewrite rules for these operators as we discussed in Chapter 7.

What is more interesting is that the relationships mentioned above allow us to define semantic rules that depend on the object model and the query model. Consider the following rules where C_i denotes the set of objects in the extent of class c_i and C_j^* denotes the deep extent of class c_j (i.e., the set of objects in the extent of c_j, as well as in the extents of all those which are subclasses of c_j):

$$
\begin{aligned}
C_1 \cap C_2 &= \phi \quad \text{if} \quad c_1 \neq c_2 \\
C_1 \cup C_2^* &= C_2^* \quad \text{if } c_1 \text{ is a subclass of } c_2 \\
(P\sigma_F \langle QSet \rangle) \cap R &\overset{c}{\Leftrightarrow} (P\sigma_F \langle QSet \rangle) \cap (R\sigma_{F'} \langle QSet \rangle) \\
&\overset{c}{\Leftrightarrow} P \cap (R\sigma_{F'} < QSet >)
\end{aligned}
$$

The first rule, for example, is true because the object model restricts each object to belong to only one class. The second rule holds because the query model permits retrieval of objects in the deep extent of the target class. Finally, the third rule relies on type consistency rules [Straube and Özsu, 1990b] for its applicability, as well as a condition (denoted by the c over the $\Leftrightarrow$) that F' is identical to F, except that each occurrence of p is replaced by r.

Since the idea of query transformation is well-known, we will not elaborate on the techniques. The above discussion only demonstrates the general idea and highlights the unique aspects that must be considered in object algebras.

Search Algorithm.

Enumerative algorithms based on dynamic programming with various optimizations are typically used for search [Selinger et al., 1979; Lee et al., 1988; Graefe and McKenna, 1993]. The combinatorial nature of enumerative search algorithms is perhaps more important in object DBMSs than in relational ones. It has been argued that if the number of joins in a query exceeds ten, enumerative search strategies become infeasible [Ioannidis and Wong, 1987]. In applications such as decision support systems, which object DBMSs are well-suited to support, it is quite common to find queries of this complexity. Furthermore, as we will address in Section 15.6.2.2, one method of executing path expressions is to represent them as explicit joins, and then use the well-known join algorithms to optimize them. If this is the case, the number of joins and other operations with join semantics in a query is quite likely to be higher than the empirical threshold of ten.

In these cases, *randomized search algorithms* (that we introduced in Chapters 7 and 8) have been suggested as alternatives to restrict the region of the search space being analyzed. Unfortunately, there has not been any study of randomized search algorithms within the context of object DBMSs. The general strategies are not likely to change, but the tuning of the parameters and the definition of the space of acceptable solutions should be expected to change. Unfortunately, the distributed versions of these algorithms are not available, and their development remains a challenge.

Cost Function.

As we have already seen, the arguments to cost functions are based on various information regarding the storage of the data. Typically, the optimizer considers the number of data items (cardinality), the size of each data item, its organization (e.g., whether there are indexes on it or not), etc. This information is readily available to the query optimizer in relational systems (through the system catalog), but may not be in object DBMSs due to encapsulation. If the query optimizer is considered "special" and allowed to look at the data structures used to implement objects, the cost functions can be specified similar to relational systems [Blakeley et al., 1993; Cluet and Delobel, 1992; Dogac et al., 1994; Orenstein et al., 1992]. Otherwise, an alternative specification must be considered.

The cost function can be defined recursively based on the algebraic processing tree. If the internal structure of objects is not visible to the query optimizer, the cost of each node (representing an algebraic operation) has to be defined. One way to define it is to have objects "reveal" their costs as part of their interface [Graefe and Maier, 1988]. In systems that uniformly implement everything as first-class objects, the cost of an operator can be a method defined on an operator implemented as a function of (a) the execution algorithm and (b) the collection over which they operate. In both cases, more abstract cost functions for operators are specified at type definition time from which the query optimizer can calculate the cost of the entire processing tree.

The definition of cost functions, especially in the approaches based on the objects revealing their costs, must be investigated further before satisfactory conclusions can be reached.

15.6.2.2 Path Expressions

Most object query languages allow queries whose predicates involve conditions on object access along reference chains. These reference chains are called *path expressions* [Zaniolo, 1983] (sometimes also referred to as *complex predicates* or *implicit joins* [Kim, 1989]). The example path expresion c.engine.manufacturer.name that we used in Section 15.4.2 retrieves the value of the name attribute of the object that is the value of the manufacturer attribute of the object that is the value of the engine attribute of object c, which was defined to be of type `Car`. It is possible to form path expressions involving attributes as well as methods. Optimizing the computation of path expressions is a problem that has received substantial attention in object-query processing.

Path expressions allow a succinct, high-level notation for expressing navigation through the object composition (aggregation) graph, which enables the formulation of predicates on values deeply nested in the structure of an object. They provide a uniform mechanism for the formulation of queries that involve object composition and inherited member functions. Path expressions may be *single-valued* or *set-valued*, and may appear in a query as part of a predicate, a target to a query (when set-valued), or part of a projection list. A path expression is single-valued if every component of a path expression is single-valued; if at least one component is set-valued, then the whole path expression is set-valued. Techniques have been developed to traverse path expressions forward and backward [Jenq et al., 1990].

The problem of optimizing path expressions spans the entire query-compilation process. During or after parsing of a user query, but before algebraic optimization, the query compiler must recognize which path expressions can potentially be optimized. This is typically achieved through *rewriting* techniques, which transform path expressions into equivalent logical algebra expressions [Cluet and Delobel, 1992]. Once path expressions are represented in algebraic form, the query optimizer explores the space of *equivalent algebraic* and execution plans, searching for one of minimal cost [Lanzelotte and Valduriez, 1991; Blakeley et al., 1993]. Finally, the optimal execution plan may involve algorithms to efficiently compute path expressions, including hash-join [Shapiro, 1986], complex-object assembly [Keller et al., 1991], or indexed scan through path indexes [Maier and Stein, 1986; Valduriez, 1987; Kemper and Moerkotte, 1990a,b].

Rewriting and Algebraic Optimization.

Consider again the path expression we used earlier: c.engine.manufacturer.name. Assume every car instance has a reference to an `Engine` object, each engine has a reference to a `Manufacturer` object, and each manufacturer instance has a `name` field. Also, assume that `Engine` and `Manufacturer` types have a corresponding type extent. The first two links of the above path may involve the retrieval of engine and manufacturer objects from disk. The third path involves only a lookup of a field within a manufacturer object. Therefore, only the first two links present opportunities for query optimization in the computation of that path. An object-query compiler needs a mechanism to distinguish these links in a path representing possible optimizations. This is typically achieved through a *rewriting* phase.

One possibility is to use a type-based rewriting technique [Cluet and Delobel, 1992]. This approach "unifies" algebraic and type-based rewriting techniques, permits factorization of common subexpressions, and supports heuristics to limit rewriting. Type information is exploited to decompose initial complex arguments of a query into a set of simpler operators, and to rewrite path expressions into joins. A similar attempt to optimizing path expressions within an algebraic framework has been devised based on joins, using an operator called *implicit join* [Lanzelotte and Valduriez, 1991]. Rules are defined to transform a series of implicit join operators into an indexed scan using a path index (see below) when it is available.

An alternative operator that has been proposed for optimizing path expressions is *materialize* (Mat) [Blakeley et al., 1993], which represents the computation of each inter-object reference (i.e., path link) explicitly. This enables a query optimizer to express the materialization of multiple components as a group using a single `Mat` operator, or individually using a Mat operator per component. Another way to think of this operator is as a "scope definition," because it brings elements of a path expression into scope so that these elements can be used in later operations or in predicate evaluation. The scoping rules are such that an object component gets into scope either by being scanned (captured using the logical Get operator in the leaves of expressions trees) or by being referenced (captured in the Mat operator). Components remain in scope until a projection discards them. The materialize operator allows a query processor to aggregate all component materializations required for the computation of a query, regardless of whether the components are needed for predicate evaluation or to produce the result of a query. The purpose of the materialize operator is to indicate to the optimizer where path expressions are used and where algebraic transformations can be applied. A number of transformation rules involving Mat are defined.

Path Indexes.

Substantial research on object query optimization has been devoted to the design of index structures to speed up the computation of path expressions [Maier and Stein, 1986; Bertino and Kim, 1989; Valduriez, 1987; Kemper and Moerkotte, 1994].

Computation of path expressions via indexes represents just one class of query-execution algorithms used in object-query optimization. In other words, efficient computation of path expressions through path indexes represents only one collection of implementation choices for algebraic operators, such as materialize and join, used to represent inter-object references. Section 15.6.3 describes a representative collection of query-execution algorithms that promise to provide a major benefit to the efficient execution of object queries. We will defer a discussion of some representative path index techniques to that section.

15.6.3 Query Execution

The relational DBMSs benefit from the close correspondence between the relational algebra operations and the access primitives of the storage system. Therefore, the generation of the execution plan for a query expression basically concerns the choice and implementation of the most efficient algorithms for executing individual algebra operators and their combinations. In object DBMSs, the issue is more complicated due to the difference in the abstraction levels of behaviorally-defined objects and their storage. Encapsulation of objects, which hides their implementation details, and the storage of methods with objects pose a challenging design problem, which can be stated as follows: "At what point in query processing should the query optimizer access information regarding the storage of objects?" One alternative is to leave this to the object manager [Straube and Özsu, 1995]. Consequently, the query-execution plan is generated from the query expression is obtained at the end of the query-rewrite step by mapping the query expression to a well-defined set of object-manager interface calls. The object-manager interface consists of a set of execution algorithms. This section reviews some of the execution algorithms that are likely to be part of future high-performance object-query execution engines.

A query-execution engine requires three basic classes of algorithms on collections of objects: *collection scan*, *indexed scan*, and *collection matching*. Collection scan is a straightforward algorithm that sequentially accesses all objects in a collection. We will not discuss this algorithm further due to its simplicity. Indexed scan allows efficient access to selected objects in a collection through an index. It is possible to use an object's field or the values returned by some method as a key to an index. Also, it is possible to define indexes on values deeply nested in the structure of an object (i.e., path indexes). In this section we mention a representative sample of path-index proposals. Set-matching algorithms take multiple collections of objects as input and produce aggregate objects related by some criteria. Join, set intersection, and assembly are examples of algorithms in this category.

15.6.3.1 Path Indexes

As indicated earlier, support for path expressions is a feature that distinguishes object queries from relational ones. Many indexing techniques designed to accelerate the computation of path expressions have been proposed [Maier and Stein, 1986; Bertino and Kim, 1989] based on the concept of join index [Valduriez, 1987].

One such path indexing technique creates an index on each class traversed by a path [Maier and Stein, 1986; Bertino and Kim, 1989]. In addition to indexes on path expressions, it is possible to define indexes on objects across their type inheritance.

Access support relations [Kemper and Moerkotte, 1994] are an alternative general technique to represent and compute path expressions. An access support relation is a data structure that stores selected path expressions. These path expressions are chosen to be the most frequently navigated ones. Studies provide initial evidence that the performance of queries executed using access support relations improves by about two orders of magnitude over queries that do not use access support relations. A system using access support relations must also consider the cost of maintaining them in the presence of updates to the underlying base relations.

15.6.3.2 Set Matching

As indicated earlier, path expressions are traversals along the composite object composition relationship. We have already seen that a possible way of executing a path expression is to transform it into a join between the source and target sets of objects. A number of different join algorithms have been proposed, such as hybrid-hash join or pointer-based hash join [Shekita and Carey, 1990]. The former uses the divide-and-conquer principle to recursively partition the two operand collections into buckets using a hash function on the join attribute. Each of these buckets may fit entirely in memory. Each pair of buckets is then joined in memory to produce the result. The pointer-based hash join is used when each object in one operand collection (call R) has a pointer to an object in the other operand collection (call S). The algorithm follows three steps, the first one being the partitioning of R in the same way as in the hybrid hash algorithm, except that it is partitioned by OID values rather than by join attribute. The set of objects S is not partitioned. In the second step, each partition R_i of R is joined with S by taking R_i and building a hash table for it in memory. The table is built by hashing each object $r \in R$ on the value of its pointer to its corresponding object in S. As a result, all R objects that reference the same page in S are grouped together in the same hash-table entry. Third, after the hash table for R_i is built, each of its entries is scanned. For each hash entry, the corresponding page in S is read, and all objects in R that reference that page are joined with the corresponding objects in S. These two algorithms are basically centralized algorithms, without any distributed counterparts. So we will not discuss them further.

An alternative method of join execution algorithm, *assembly* [Keller et al., 1991], is a generalization of the pointer-based hash-join algorithm for the case when a multi-way join needs to be computed. Assembly has been proposed as an additional object algebra operator. This operation efficiently assembles the fragments of objects' states required for a particular processing step, and returns them as a complex object in memory. It translates the disk representations of complex objects into readily traversable memory representations.

Assembling a complex object rooted at objects of type R containing object components of types S, U, and T, is analogous to computing a four-way join of these sets.

There is a difference between assembly and n-way pointer joins in that assembly does not need the entire collection of root objects to be scanned before producing a single result.

Instead of assembling a single complex object at a time, the assembly operator assembles a *window*, of size W, of complex objects simultaneously. As soon as any of these complex objects becomes assembled and passed up the query-execution tree, the assembly operator retrieves another one to work on. Using a window of complex objects increases the pool size of unresolved references and results in more options for optimization of disk accesses. Due to the randomness with which references are resolved, the assembly operator delivers assembled objects in random order up the query execution tree. This behavior is correct in set-oriented query processing, but may not be for other collection types, such as lists.

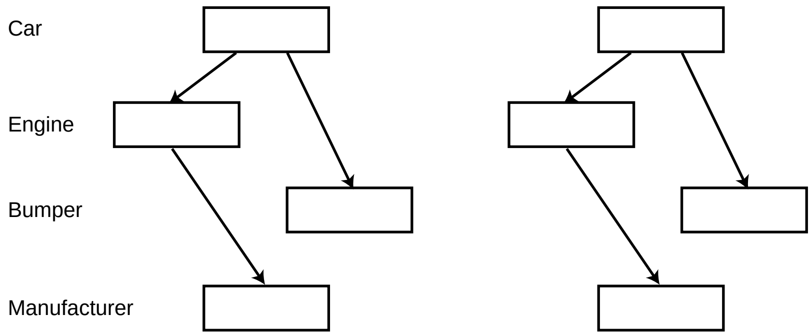


Fig. 15.3 Two Assembled Complex Objects

Example 15.8. Consider the example given in Figure 15.3, which assembles a set of Car objects. The boxes in the figure represent instances of types indicated at the left, and the edges denote the composition relationships (e.g., there is an attribute of every object of type `Car` that points to an object of type `Engine`). Suppose that assembly is using a window of size 2. The assembly operator begins by filling the window with two (since $W = 2$) `Car` object references from the set (Figure 15.4a). The assembly operator begins by choosing among the current outstanding references, say $C1$. After resolving (fetching) $C1$, two new unresolved references are added to the list (Figure 15.4b). Resolving $C2$ results in two more references added to the list (Figure 15.4c), and so on until the first complex object is assembled (Figure 15.4g). At this point, the assembled object is passed up the query-execution tree, freeing some window space. A new `Car` object reference, $C3$, is added to the list and then resolved, bringing two new references $E3$, $B3$ (Figure 15.4h). ♦

The objective of the assembly algorithm is to simultaneously assemble a window of complex objects. At each point in the algorithm, the outstanding reference that optimizes disk accesses is chosen. There are different orders, or schedules, in which references may be resolved, such as depth-first, breath-first, and elevator. Performance

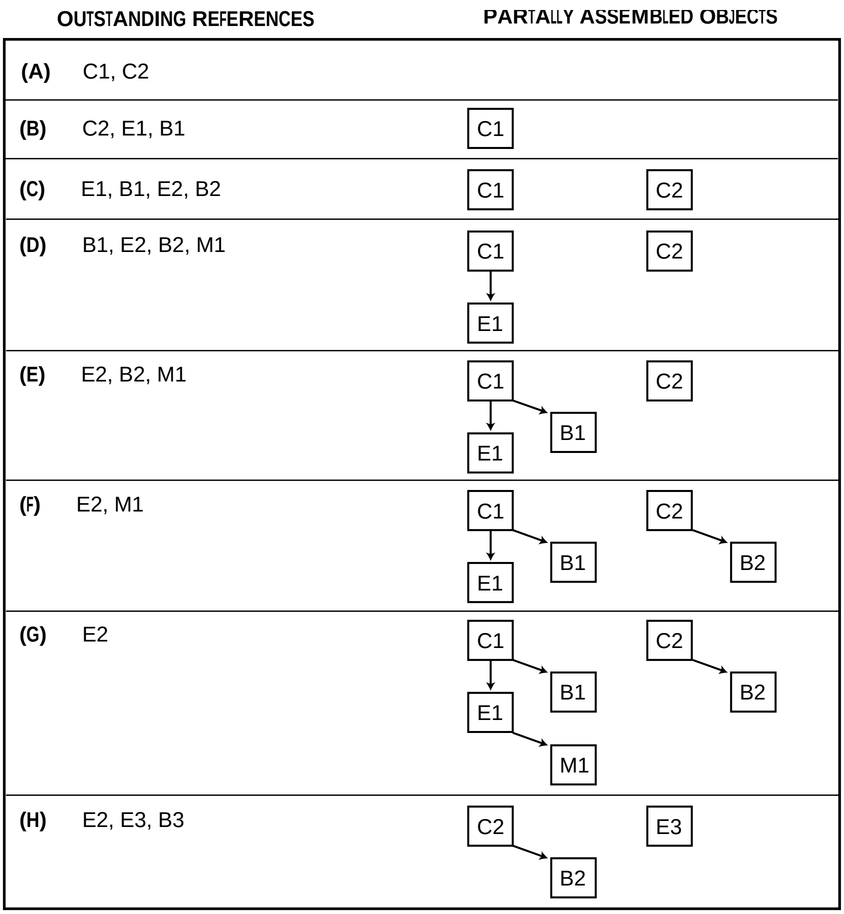


Fig. 15.4 An Assembly Example

results indicate that elevator outperforms depth-first and breath-first under several data-clustering situations [Keller et al., 1991].

A number of possibilities exist in implementing a distributed version of this operation [Maier et al., 1994]. One strategy involves shipping all data to a central site for processing. This is straightforward to implement, but could be inefficient in general. A second strategy involves doing simple operations (e.g., selections, local assembly) at remote sites, then shipping all data to a central site for final assembly. This strategy also requires fairly simple control, since all communication occurs through the central site. The third strategy is significantly more complicated: perform complex operations (e.g., joins, complete assembly of remote objects) at remote sites, then ship the results to the central site for final assembly. A distributed object DBMS may include all or some of these strategies.

15.7 Transaction Management

Transaction management in *distributed* object DBMSs have not been studied except in relation to the cashing problem discussed earlier. However, transactions on objects raise a number of interesting issues, and their execution in a distributed environment can be quite challenging. This is an area that clearly requires more work. In this section we will briefly discuss the particular problems that arise in extending the transaction concept to object DBMSs.

Most object DBMSs maintain page level locks for concurrency control and support the traditional flat transaction model. It has been argued that the traditional flat transaction model would not meet the requirements of the advanced application domains that object data management technology would serve. Some of the considerations are that transactions in these domains are longer in duration, requiring interactions with the user or the application program during their execution. In the case of object systems, transactions do not consist of simple read/write operations, necessitating, instead, synchronization algorithms that deal with complex operations on abstract (and possibly complex) objects. In some application domains, the fundamental transaction synchronization paradigm based on competition among transactions for access to resources must change to one of cooperation among transactions in accomplishing a common task. This is the case, for example, in cooperative work environments.

The more important requirements for transaction management in object DBMSs can be listed as follows [Buchmann et al., 1982; Kaiser, 1989; Martin and Pedersen, 1994]:

1. Conventional transaction managers synchronize simple Read and Write operations. However, their counterparts for object DBMSs must be able to deal with *abstract operations*. It may even be possible to improve concurrency by using semantic knowledge about the objects and their abstract operations.
2. Conventional transactions access "flat" objects (e.g., pages, tuples), whereas transactions in object DBMSs require synchronization of access to composite and complex objects. Synchronization of access to such objects requires synchronization of access to the component objects.
3. Some applications supported by object DBMSs have different database access patterns than conventional database applications, where the access is competitive (e.g., two users accessing the same bank account). Instead, sharing is more cooperative, as in the case of, for example, multiple users accessing and working on the same design document. In this case, user accesses must be synchronized, but users are willing to cooperate rather than compete for access to shared objects.
4. These applications require the support of *long-running activities* spanning hours, days or even weeks (e.g., when working on a design object). Therefore, the transaction mechanism must support the sharing of partial results. Furthermore, to avoid the failure of a partial task jeopardizing a long activity, it is necessary to distinguish between those activities that are essential for

the completion of a transaction and those that are not, and to provide for alternative actions in case the primary activity fails.

5. It has been argued that many of these applications would benefit from *active capabilities* for timely response to events and changes in the environment. This new database paradigm requires the monitoring of events and the execution of system-triggered activities within running transactions.

These requirements point to a need to extend the traditional transaction management functions in order to capture application and data semantics, and to a need to relax isolation properties. This, in turn, requires revisiting every aspect of transaction management that we discussed in Chapters 10–12.

15.7.1 Correctness Criteria

In Chapter 11, we introduced serializability as the fundamental correctness criteria for concurrent execution of database transactions. There are a number of different ways in which serializability can be defined, even though we did not elaborate on this point before. These differences are based on how a *conflict* is defined. We will concentrate on three alternatives: *commutativity* [Weihl, 1988, 1989; Fekete et al., 1989], *invalidation* [Herlihy, 1990], and *recoverability* [Badrinath and Ramamritham, 1987].

15.7.1.1 Commutativity

Commutativity states that two operations conflict if the results of different serial executions of these operations are not equivalent. We had briefly introduced commutativity within the context of ordered-shared locks in Chapter 11 (see Figure 11.8). The traditional conflict definition discussed in Chapter 11 is a special case. Consider the simple operations $R(x)$ and $W(x)$. If nothing is known about the abstract semantics of the Read and Write operations or the object x upon which they operate, it has to be accepted that a $R(x)$ **following** a $W(x)$ does not retrieve the same value as it would **prior** to the $W(x)$. Therefore, a Write operation always conflicts with other Read or Write operations. The conflict table (or the compatibility matrix) given in Figure 11.5 for Read and Write operations is, in fact, derived from the commutativity relationship between these two operations. This table was called the compatibility matrix in Chapter 11, since two operations that do not conflict are said to be compatible. Since this type of commutativity relies only on syntactic information about operations (i.e., that they are Read and Write), we call this *syntactic commutativity* [Buchmann et al., 1982].

In Figure 11.5, Read and Write operations and Write and Write operations do not commute. Therefore, they conflict, and serializability maintains that either all conflicting operations of transaction T_i precede all conflicting operations of T_k, or vice versa.

If the semantics of the operations are taken into account, however, it may be possible to provide a more relaxed definition of conflict. Specifically, some concurrent executions of Write-Write and Read-Write may be considered non-conflicting. *Semantic commutativity* (e.g., [Weihl, 1988, 1989]) makes use of the semantics of operations and their termination conditions.

Example 15.9. Consider, for example, an abstract data type **set** and three operations defined on it: Insert and Delete, which correspond to a Write, and Member, which tests for membership and corresponds to a Read. Due to the semantics of these operations, two Insert operations on an instance of set type would commute, allowing them to be executed concurrently. The commutativity of Insert with Member and the commutativity of Delete with Member depends upon whether or not they reference the same argument and their results[7]. ♦

It is also possible to define commutativity with reference to the database state. In this case, it is usually possible to permit more operations to commute.

Example 15.10. In Example 15.7, we indicated that an Insert and a Member would commute if they do not refer to the same argument. However, if the set already contains the referred element, these two operations would commute even if their arguments are the same. ♦

15.7.1.2 Invalidation

Invalidation [Herlihy, 1990] defines a conflict between two operations not on the basis of whether they commute or not, but according to whether or not the execution of one invalidates the other. An operation P invalidates another operation Q if there are two histories H_1 and H_2 such that $H_1 \bullet P \bullet H_2$ and $H_1 \bullet H_2 \bullet Q$ are legal, but $H_1 \bullet P \bullet H_2 \bullet Q$ is not. In this context, a *legal history* represents a correct history for the set object and is determined according to its semantics. Accordingly, an *invalidated-by* relation is defined as consisting of all operation pairs (P, Q) such that P invalidates Q. The invalidated-by relation establishes the conflict relation that forms the basis of establishing serializability. Considering the Set example, an Insert cannot be invalidated by any other operation, but a Member can be invalidated by a Delete if their arguments are the same.

[7] Depending upon the operation, the result may either be a flag that indicates whether the operation was successful (for example, the result of Insert may be "OK") or the value that the operation returns (as in the case of a Read).

15.7.1.3 Recoverability

Recoverability [Badrinath and Ramamritham, 1987] is another conflict relation that has been defined to determine serializable histories[8]. Intuitively, an operation P is said to be *recoverable with respect to* operation Q if the value returned by P is independent of whether Q executed before P or not. The conflict relation established on the basis of recoverability seems to be identical to that established by invalidation. However, this observation is based on only a few examples, and there is no formal proof of this equivalence. In fact, the absence of a formal theory to reason about these conflict relations is a serious deficiency that must be addressed.

15.7.2 Transaction Models and Object Structures

In Chapter 10, we considered a number of transaction models ranging from flat transactions to workflow systems. All of these alternatives access simple database objects (sets of tuples or a physical page). In the case of object databases, however, the database objects are not simple; they can be objects with state and properties, they can be complex objects, or even active objects (i.e., objects that are capable of responding to events by triggering the execution of actions when certain conditions are satisfied). The complications added by the complexity of objects is significant, as we highlight in subsequent sections.

15.7.3 Transactions Management in Object DBMSs

Transaction management techniques that are developed for object DBMSs need to take into consideration the complications we discussed earlier: they need to employ more sophisticated correctness criteria that take into account method semantics, they need to consider the object structure, they need to be cognizant of the composition and inheritance relationships. In addition to these structures, object DBMSs store methods together with data. Synchronization of shared access to objects must take into account method executions. In particular, transactions invoke methods which may, in turn, invoke other methods. Thus, even if the transaction model is flat, the execution of these transactions may be dynamically nested.

[8] Recoverability as used in [Badrinath and Ramamritham, 1987] is different from the notion of recoverability as we defined it in Chapter 12 and as found in [Bernstein et al., 1987] and [Hadzilacos, 1988].

15.7.3.1 Synchronizing Access to Objects

The inherent nesting in method invocations can be used to develop algorithms based on the well-known nested 2PL and nested timestamp ordering algorithms [Hadzilacos and Hadzilacos, 1991]. In the process, intra-object parallelism may be exploited to improve concurrency. In other words, attributes of an object can be modeled as data elements in the database, whereas the methods are modeled as transactions enabling multiple invocations of an object's methods to be active simultaneously. This can provide more concurrency if special intra-object synchronization protocols can be devised that maintain the compatibility of synchronization decisions at each object.

Consequently, a method execution (modeled as a transaction) on an object consists of *local steps*, which correspond to the execution of local operations together with the results that are returned, and *method steps*, which are the method invocations together with the return values. A local operation is an atomic operation (such as Read, Write, Increment) that affects the object's variables. A method execution defines the partial order among these steps in the usual manner.

One of the fundamental directions of this work is to provide total freedom to objects in how they achieve intra-object synchronization. The only requirement is that they be "correct" executions, which, in this case, means that they should be serializable based on commutativity. As a result of the delegation of intra-object synchronization to individual objects, the concurrency control algorithm concentrates on inter-object synchronization.

An alternative approach is multigranularity locking [Garza and Kim, 1988; Cart and Ferrie, 1990]. Multigranularity locking defines a hierarchy of lockable database granules (thus the name "granularity hierarchy") as depicted in Figure 15.5. In relational DBMSs, files correspond to relations and records correspond to tuples. In object DBMSs, the correspondence is with classes and instance objects, respectively. The advantage of this hierarchy is that it addresses the tradeoff between coarse granularity locking and fine granularity locking. Coarse granularity locking (at the file level and above) has low locking overhead, since a small number of locks are set, but it significantly reduces concurrency. The reverse is true for fine granularity locking.

The main idea behind multigranularity locking is that a transaction that locks at a coarse granularity implicitly locks all the corresponding objects of finer granularities. For example, explicit locking at the file level involves implicit locking of all the records in that file. To achieve this, two more lock types in addition to shared (S) and exclusive (X) are defined: *intention* (or *implicit*) *shared* (IS) and *intention* (or *implicit*) *exclusive* (IX). A transaction that wants to set an S or an IS lock on an object has to first set IS or IX locks on its ancestors (i.e., related objects of coarser granularity). Similarly, a transaction that wants to set an X or an IX lock on an object must set IX locks on all of its ancestors. Intention locks cannot be released on an object if the descendants of that object are currently locked.

One additional complication arises when a transaction wants to read an object at some granularity and modify some of its objects at a finer granularity. In this case, both an S lock and an IX lock must be set on that object. For example, a transaction

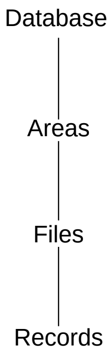


Fig. 15.5 Multiple Granularities

may read a file and update some records in that file (similarly, a transaction in object DBMSs may want to read the class definition and update some of the instance objects belonging to that class). To deal with these cases, a *shared intention exclusive* (SIX) lock is introduced, which is equivalent to holding an S and an IX lock on that object. The lock compatibility matrix for multigranularity locking is shown in Figure 15.6.

A possible granularity hierarchy is shown in Figure 15.7. The lock modes that are supported and their compatibilities are exactly those given in Figure 15.6. Instance objects are locked only in S or X mode, while class objects can be locked in all five modes. The interpretation of these locks on class objects is as follows:

- S mode: Class definition is locked in S mode, and all its instances are implicitly locked in S mode. This prevents another transaction from updating the instances.

	S	X	IS	IX	SIX
S	+	-	+	-	-
X	-	-	-	-	-
IS	+	-	+	+	+
IX	-	-	+	+	-
SIX	-	-	+	-	-

Fig. 15.6 Compatibility Table for Multigranularity Locking

- X mode: Class definition is locked in X mode, and all its instances are implicitly locked in X mode. Therefore, the class definition and all instances of the class may be read or updated.

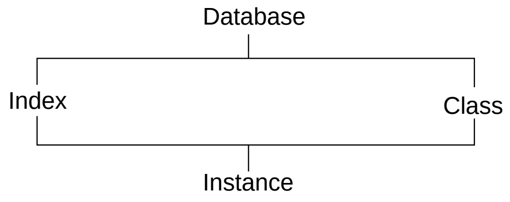


Fig. 15.7 Granularity Hierarchy

- IS mode: Class definition is locked in IS mode, and the instances are to be locked in S mode as necessary.
- IX mode: Class definition is locked in IX mode, and the instances will be locked in either S or X mode as necessary.
- SIX mode: Class definition is locked in S mode, and all the instances are implicitly locked in S mode. Those instances that are to be updated are explicitly locked in X mode as the transaction updates them.

15.7.3.2 Management of Class Lattice

One of the important requirements of object DBMSs is dynamic schema evolution. Consequently, systems must deal with transactions that access schema objects (i.e., types, classes, etc.), as well as instance objects. The existence of schema change operations intermixed with regular queries and transactions, as well as the (multiple) inheritance relationship defined among classes, complicates the picture. First, a query/transaction may not only access instances of a class, but may also access instances of subclasses of that class (i.e., *deep extent*). Second, in a composite object, the domain of an attribute is itself a class. So accessing an attribute of a class may involve accessing the objects in the sublattice rooted at the domain class of that attribute.

One way to deal with these two problems is, again, by using multigranularity locking. The straightforward extension of multigranularity locking where the accessed class and all its subclasses are locked in the appropriate mode does not work very well. This approach is inefficient when classes close to the root are accessed, since it involves too many locks. The problem may be overcome by introducing *read-lattice* (R) and *write-lattice* (W) lock modes, which not only lock the target class in S or X modes, respectively, but also implicitly lock all subclasses of that class in S and X modes, respectively. However, this solution does not work with multiple inheritance (which is the third problem).

The problem with multiple inheritance is that a class with multiple supertypes may be implicitly locked in incompatible modes by two transactions that place R and W locks on different superclasses. Since the locks on the common class are implicit, there is no way of recognizing that there is already a lock on the class. Thus, it is necessary to check the superclasses of a class that is being locked. This can be

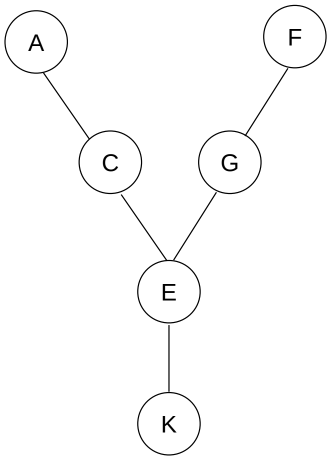


Fig. 15.8 An Example Class Lattice

handled by placing *explicit* locks, rather than implicit ones, on subclasses. Consider the type lattice of Figure 15.8, which is simplified from [Garza and Kim, 1988]. If transaction T_1 sets an IR lock on class A and an R lock on C, it also sets an explicit R lock on E. When another transaction T_2 places an IW lock on F and a W lock on G, it will attempt to place an explicit W lock on E. However, since there is already an R lock on E, this request will be rejected.

An alternative to setting explicit locks is to set locks at a finer granularity, uses ordered sharing, as discussed in Chapter 11 [Agrawal and El-Abbadi, 1994]. In a sense, the algorithm is an extension of Weihl's commutativity-based approach to object DBMSs using a nested transaction model.

Classes are modeled as objects in the system similar to reflective systems that represent schema objects as first-class objects. Consequently, methods can be defined that operate on class objects: $add(m)$ to add method m to the class, $del(m)$ to delete method m from the class, $rep(m)$ to replace the implementation of method m with another one, and $use(m)$ to execute method m. Similarly, atomic operations are defined for accessing attributes of a class. These are identical to the method operations with the appropriate change in semantics to reflect attribute access. The interesting point to note here is that the definition of the $use(a)$ operation for attribute a indicates that the access of a transaction to attribute a within a method execution is through the *use* operation. This requires that each method explicitly list all the attributes that it accesses. Thus, the following is the sequence of steps that are followed by a transaction, T, in executing a method m:

1. Transaction T issues operation $use(m)$.
2. For each attribute a that is accessed by method m, T issues operation $use(a)$.
3. Transaction T invokes method m.

Commutativity tables are defined for the method and attribute operations. Based on the commutativity tables, ordered sharing lock tables for each atomic operation are determined (see Figure 11.8). Specifically, a lock for an atomic operation p has a shared relationship with all the locks associated with operations with which p has a non-conflicting relationship, whereas it has an ordered shared relationship with respect to all the locks associated with operations with which p has a conflicting relation.

Based on these lock tables, a nested 2PL locking algorithm is used with the following considerations:

1. Transactions observe the strict 2PL rule and hold on to their locks until termination.
2. When a transaction aborts, it releases all of its locks.
3. The termination of a transaction awaits the termination of its children (closed nesting semantics). When a transaction commits, its locks are inherited by its parent.
4. *Ordered commitment rule.* Given two transactions T_i and T_j such that T_i is *waiting for* T_j, T_i cannot commit its operations on any object until T_j terminates (commits or aborts). T_i is said to be *waiting-for* T_j if:
   - T_i is not the root of the nested transaction and T_i was granted a lock in ordered shared relationship with respect to a lock held by T_j on an object such that T_j is a descendent of the parent of T_i; or
   - T_i is the root of the nested transaction and T_i holds a lock (that it has inherited or it was granted) on an object in ordered shared relationship with respect to a lock held by T_j or its descendants.

15.7.3.3 Management of Composition (Aggregation) Graph

Studies dealing with the composition graph are more prevalent. The requirement for object DBMSs to model composite objects in an efficient manner has resulted in considerable interest in this problem.

One approach is based on multigranularity locking where one can lock a composite object and all the classes of the component objects. This is clearly unacceptable, since it involves locking the entire composite object hierarchy, thereby restricting performance significantly. An alternative is to lock the component object instances within a composite object. In this case, it is necessary to chase all the references and lock all those objects. This is quite cumbersome, since it involves locking so many objects.

The problem is that the multigranularity locking protocol does not recognize the composite object as one lockable unit. To overcome this problem, three new lock modes are introduced: ISO, IXO, and SIXO, corresponding to the IS, IX, and SIX modes, respectively. These lock modes are used for locking component classes of

	S	X	IS	IX	SIX	ISO	IXO	SIXO
S	+	-	+	-	-	+	-	-
X	-	-	-	-	-	-	-	-
IS	+	-	+	+	+	+	-	-
IX	-	-	+	+	-	-	-	-
SIX	-	-	+	-	-	-	-	-
ISO	+	-	+	+	-	+	+	+
IXO	-	-	-	-	-	+	+	-
SIXO	N	N	N	N	N	Y	N	N

Fig. 15.9 Compatibility Matrix for Composite Objects

a composite object. The compatibility of these modes is shown in Figure 15.9. The protocol is then as follows: to lock a composite object, the root class is locked in X, IS, IX, or SIX mode, and each of the component classes of the composite object hierarchy is locked in the X, ISO, IXO, and SIXO mode, respectively.

Another approach extends multigranularity locking by replacing the single static lock graph with a hierarchy of graphs associated with each type and query [Herrmann et al., 1990]. There is a "general lock graph" that controls the entire process (Figure 15.10). The smallest lockable units are called *basic lockable units* (BLU). A number of BLUs can make up a *homogeneous lockable unit* (HoLU), which consists of data of the same type. Similarly, they can make up a *heterogeneous lockable unit* (HeLU), which is composed of objects of different types. HeLUs can contain other HeLUs or HoLUs, indicating that component objects do not all have to be atomic. Similarly, HoLUs can consist of other HoLUs or HeLUs, as long as they are of the same type. The separation between HoLUs and HeLUs is meant to optimize lock requests. For example, a set of lists of integers is, from the viewpoint of lock managers, treated as a HoLU composed of HoLUs, which, in turn, consist of BLUs. As a result, it is possible to lock the whole set, exactly one of the lists, or even just one integer.

At type definition time, an object-specific lock graph is created that obeys the general lock graph. As a third component, a query-specific lock graph is generated during query (transaction) analysis. During the execution of the query (transaction), the query-specific lock graph is used to request locks from the lock manager, which uses the object-specific lock graph to make the decision. The lock modes used are the standard ones (i.e., IS, IX, S, X).

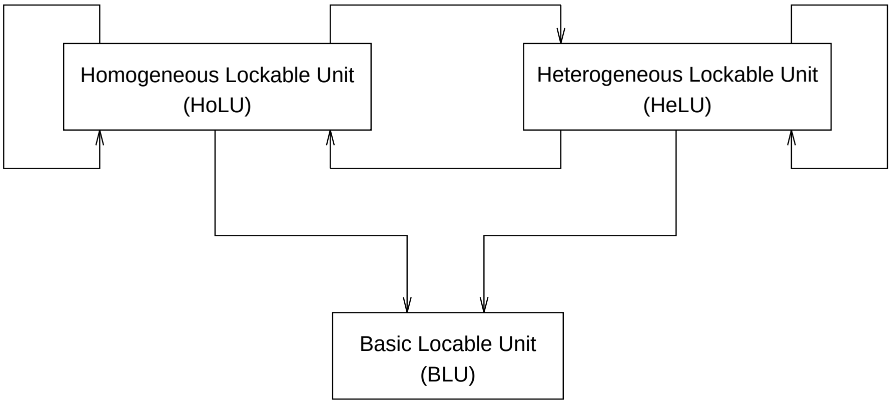

Homogeneous Lockable Unit (HoLU)
Heterogeneous Lockable Unit (HeLU)
Basic Locable Unit (BLU)

Fig. 15.10 General Lock Graph

Badrinath and Ramamritham [1987] discuss an alternative to dealing with composite object hierarchy based on commutativity. A number of different operations are defined on the aggregation graph:

1. Examine the contents of a vertex (which is a class).
2. Examine an edge (composed-of relationship).
3. Insert a vertex and the associated edge.
4. Delete a vertex and the associated edge.
5. Insert an edge.

Note that some of these operations (1 and 2) correspond to existing object operators, while others (3—5) represent schema operations.

Based on these operations, an *affected-set* can be defined for granularity graphs to form the basis for determining which operations can execute concurrently. The affected-set of a granularity graph consists of the union of:

- *edge-set*, which is the set of pairs (e,a) where e is an edge and a is an operation affecting e and can be one of *insert*, *delete*, *examine*; and
- *vertex-set*, which is the set of pairs (v,a), where v is a vertex and a is an operation affecting v and can be one of *insert*, *delete*, *examine*, or *modify*.

Using the affected-set generated by two transactions T_i and T_j of an aggregation graph, one may define whether T_i and T_j can execute concurrently or not. Commutativity is used as the basis of the conflict relation. Thus, two transactions T_i and T_j commute on object K if $\textit{affected-set}(T_i) \cap_K \textit{affected-set}(T_j) = \phi$.

These protocols synchronize on the basis of objects, not operations on objects. It may be possible to improve concurrency by developing techniques that synchronize operation invocations rather than locking entire objects.

Another semantics-based approach due to Muth et al. [1993] has the following distinguishing characteristics:

1. Access to component objects are permitted without going through a hierarchy of objects (i.e., no multigranularity locking).
2. The semantics of operations are taken into consideration by a priori specification of method commutativities[9].
3. Methods invoked by a transaction can themselves invoke other methods. This results in a (dynamic) nested transaction execution, even if the transaction is syntactically flat.

The transaction model used to support (3) is open nesting, specifically multilevel transactions as described in Chapter 10. The restrictions imposed on the dynamic transaction nesting are:

- All pairs (p,g) of potentially conflicting operations on the same object have the same depth in their invocation trees; and
- For each pair (f',g') of ancestors of f and g whose depth of invocation trees are the same, f' and g' operate on the same object.

With these restrictions, the algorithm is quite straightforward. A semantic lock is associated with each method, and a commutativity table defines whether or not the various semantic locks are compatible. Transactions acquire these semantic locks before the invocation of methods, and they are released at the end of the execution of a subtransaction (method), exposing their results to others. However, the parents of committed subtransactions have a higher-level semantic lock, which restricts the results of committed subtransactions only to those that commute with the root of the subtransaction. This requires the definition of a semantic conflict test, which operates on the invocation hierarchies using the commutativity tables.

An important complication arises with respect to the two conditions outlined above. It is not reasonable to restrict the applicability of the protocol to only those for which those conditions hold. What has been proposed to resolve the difficulty is to give up some of the openness and convert the locks that were to be released at the end of a subtransaction into *retained locks* held by the parent. A number of conditions under which retained locks can be discarded for additional concurrency.

A very similar, but more restrictive, approach is discussed by Weikum and Hasse [1993]. The multilevel transaction model is used, but restricted to only two levels: the object level and the underlying page level. Therefore, the dynamic nesting that occurs when transactions invoke methods that invoke other methods is not considered. The similarity with the above work is that page level locks are released at the end of the subtransaction, whereas the object level locks (which are semantically richer) are retained until the transaction terminates.

[9] The commutativity test employed in this study is state-independent. It takes into account the actual parameters of operations, but not the states. This is in contrast to Weihl's work [Weihl, 1988].

In both of the above approaches [Muth et al., 1993; Weikum and Hasse, 1993], recovery cannot be performed by page-level state-oriented protocols. Since subtransactions release their locks and make their results visible, compensating transactions must be run to "undo" actions of committed subtransactions.

15.7.4 Transactions as Objects

One important characteristic of relational data model is its lack of a clear update semantics. The model, as it was originally defined, clearly spells out how the data in a relational database is to be retrieved (by means of the relational algebra operators), but does not specify what it really means to update the database. The consequence is that the consistency definitions and the transaction management techniques are orthogonal to the data model. It is possible – and indeed it is common – to apply the same techniques to non-relational DBMSs, or even to non-DBMS storage systems.

The independence of the developed techniques from the data model may be considered an advantage, since the effort can be amortized over a number of different applications. Indeed, the existing transaction management work on object DBMSs have exploited this independence by porting the well-known techniques over to the new system structures. During this porting process, the peculiarities of object DBMSs, such as class (type) lattice structures, composite objects and object groupings (class extents) are considered, but the techniques are essentially the same.

It may be argued that in object DBMSs, it is not only desirable but indeed essential to model update semantics within the object model. The arguments are as follows:

1. In object DBMSs, what is stored are not only data, but operations on data (which are called methods, behaviors, operations in various object models). Queries that access an object database refer to these operations as part of their predicates. In other words, the execution of these queries invokes various operations defined on the classes (types). To guarantee the safety of the query expressions, existing query processing approaches restrict these operations to be side-effect free, in effect disallowing them to update the database. This is a severe restriction that should be relaxed by the incorporation of update semantics into the query safety definitions.
2. As we discussed in Section 15.7.3, transactions in object DBMSs affect the class (type) lattices. Thus, there is a direct relationship between dynamic schema evolution and transaction management. Many of the techniques that we discussed employ locking on this lattice to accommodate these changes. However, locks (even multi-granularity locks) severely restrict concurrency. A definition of what it means to update a database, and a definition of conflicts based on this definition of update semantics, would allow more concurrency. It is interesting to note again the relationship between changes to the class (type) lattice and query processing. In the absence of a clear definition of update semantics and its incorporation into the query processing methodology,

most of the current query processors assume that the database schema (i.e., the class (type) lattice) is static during the execution of a query.

3. There are a few object models (e.g., OODAPLEX [Dayal, 1989] and TIGUKAT [Özsu et al., 1995a]) that treat all system entities as objects. Following this approach, it is only natural to model transactions as objects. However, since transactions are basically constructs that change the state of the database, their effects on the database must be clearly specified.
   Within this context, it should also be noted that the application domains that require the services of object DBMSs tend to have somewhat different transaction management requirements, both in terms of transaction models and consistency constraints. Modeling transactions as objects enables the application of the well-known object techniques of specialization and subtyping to create various different types of TMSs. This gives the system extensibility.
4. Some of the requirements require rule support and active database capabilities. Rules themselves execute as transactions, which may spawn other transactions. It has been argued that rules should be modeled as objects [Dayal et al., 1988]. If that is the case, then certainly transactions should be modeled as objects too.

As a result of these points, it seems reasonable to argue for an approach to transaction management systems that is quite different from what has been done up to this point. This is a topic of some research potential.

15.8 Conclusion

In this chapter we considered the effect of object technology on database management and focused on the distribution aspects when possible. Research into object technologies was widespread in the 1980's and the first half of 1990's. Interest in the topic died down primarily as a result of two factors. The first was that object DBMSs were claimed to be replacements for relational ones, rather than specialized systems that better fit certain application requirements. The object DBMSs, however, were not able to deliver the performance of relational systems for those applications that really fit the relational model well. Consequently, they were easy targets for the relational proponents, which is the second factor. The relational vendors adopted many of the techniques developed for object DBMSs into their products and released "object-relational DBMSs", as noted earlier, allowing them to claim that there is no reason for a new class of systems. The object extensions to relational DBMSs work with varying degrees of success. They allow the attributes to be structured, allowing non-normalized relations. They are also extensible by enabling the insertion of new data types into the system by means of *data blades*, *cartridges*, or *extenders* (each commercial system uses a different name). However, this extensibility is limited, as it requires significant effort to write a data blade/cartridge/extender, and their robustness is a considerable issue.

In recent years, there has been a re-emergence of object technology. This is spurred by the recognition of the advantages of these systems in particular applications that are gaining importance. For example, the DOM interface of XML, the Java Data Objects (JDO) API are all object-oriented and they are crucial technologies. JDO has been critically important in resolving the mapping problems between Java Enterprice Edition (J2EE) and relational systems. Object-oriented middleware architectures such as CORBA Siegel [1996] have not been as influential as they could be in their first incarnation, but they have been demonstrated to contribute to database interoperability [Dogac et al., 1998a], and there is continuing work in improving them.

15.9 Bibliographic Notes

There are a number of good books on object DBMSs such as [Kemper and Moerkotte, 1994; Bertino and Martino, 1993; Cattell, 1994] and [Dogac et al., 1994]. An early collection of readings in object DBMSs is [Zdonik and Maier, 1990]. In addition, object DBMS concepts are discussed in [Kim and Lochovsky, 1989; Kim, 1994]. These are, unfortunately, somewhat dated. [Orfali et al., 1996] is considered the classical book on distributed objects, but the emphasis is mostly on the distributed object platforms (CORBA and COM), not on the fundamental DBMS functionality. Considerable work has been done on the formalization of object models, some of which are discussed in [Abadi and Cardelli, 1996; Maier, 1986; Chen and Warren, 1989; Kifer and Wu, 1993; Kifer et al., 1995; Abiteboul and Kanellakis, 1998b; Guerrini et al., 1998].

Our discussion of the architectural issues is mostly based on [Özsu et al., 1994a] but largely extended. The object distribution design issues are discussed in significant more detail in [Ezeife and Barker, 1995], [Bellatreche et al., 2000a], and [Bellatreche et al., 2000b]. A formal model for distributed objects is given in [Abiteboul and dos Santos, 1995]. The query processing and optimization section is based on [Özsu and Blakeley, 1994] and the transaction management issues are from [Özsu, 1994]. Related work on indexing techniques for query optimization have been discussed in [Bertino et al., 1997; Kim and Lochovsky, 1989]. Several techniques for distributed garbage collection have been classified in a survey article by Plainfossé and Shapiro [1995]. These sources contain more detail than can be covered in one chapter. Object-relational DBMSs are discussed in detail in [Stonebraker and Brown, 1999] and [Date and Darwen, 1998].

Exercises

Problem 15.1. Explain the mechanisms used to support encapsulation in distributed object DBMSs. In particular:

(a) Describe how the encapsulation is hidden from the end users when both the objects and the methods are distributed.
(b) How does a distributed object DBMS present a single global schema to end users? How is this different from supporting fragmentation transparency in relational database systems?

Problem 15.2. List the new data distribution problems that arise in object DBMSs, that are not present in relational DBMSs, with respect to fragmentation, migration and replication.

Problem 15.3 (**). Partitioning of object databases has the premise of reducing the irrelevant data access for user applications. Develop a cost model to execute queries on unpartitioned object databases, and horizontally or vertically partitioned object databases. Use your cost model to illustrate the scenarios under which partitioning does in fact reduce the irrelevant data access.

Problem 15.4. Show the relationship between clustering and partitioning. Illustrate how clustering can deteriorate/improve the performance of queries on a partitioned object database system.

Problem 15.5. Why do client-server object DBMSs primarily employ data shipping architecture while relational DBMSs emply function shipping?

Problem 15.6. Discuss the strengths and weaknesses of page and object servers with respect to data transfer, buffer management, cache consistency, and pointer swizzling mechanims.

Problem 15.7. What is the difference between caching information at the clients and data replication?

Problem 15.8 (*). A new class of applications that object DBMSs support are interactive and deal with large objects (e.g., interactive multimedia systems). Which one of the cache consistency algorithms presented in this chapter are suitable for this class of applications operating across wide area networks?

Problem 15.9 (**). Hardware and software pointer swizzling mechanisms have complementary strengths and weaknesses. Propose a hybrid pointer swizzling mechanism that incorporates the strengths of both.

Problem 15.10 (**). Explain how derived horizontal fragmentation can be exploited to facilitate efficient path queries in distributed object DBMSs. Give examples.

Problem 15.11 (**). Give some heuristics that an object DBMS query optimizer that accepts OQL queries may use to determine how to decompose a query so that parts can be function shipped and other parts have to be executed at the originating client by data shipping.

Problem 15.12 (**). Three alternative ways of performing *distributed* complex object assembly are discussed in this chapter. Give an algorithm for the alternative where complex operations, such as joins and complete assembly of remote objects, are performed at remote sites and the partial results are shipped to the central site for final assembly.

Problem 15.13 (*). Consider the airline reservation example of Chapter 10. Define a Reservation class (type) and give the forward and backward commutativity matrixes for it.

Chapter 16
Peer-to-Peer Data Management

In this chapter, we discuss the data management issues in the "modern" peer-to-peer (P2P) data management systems. We intentionally use the phrase "modern" to differentiate these from the early P2P systems that were common prior to client/server computing. As indicated in Chapter 1, early work on distributed DBMSs had primarily focused on P2P architectures where there was no differentiation between the functionality of each site in the system. So, in one sense, P2P data management is quite old – if one simply interprets P2P to mean that there are no identifiable "servers" and "clients" in the system. However, the "modern" P2P systems go beyond this simple characterization and differ from the old systems that are referred to by the same name in a number of important ways, as mentioned in Chapter 1.

The first difference is the massive distribution in current systems. While the early systems focused on a few (perhaps at most tens of) sites, current systems consider thousands of sites. Furthermore, these sites are geographically very distributed, with possible clusters forming at certain locations.

The second is the inherent heterogeneity of every aspect of the sites and their autonomy. While this has always been a concern of distributed databases, coupled with massive distribution, site heterogeneity and autonomy take on added significance, disallowing some of the approaches from consideration.

The third major difference is the considerable volatility of these systems. Distributed DBMSs are well-controlled environments, where additions of new sites or the removal of existing sites is done very carefully and rarely. In modern P2P systems, the sites are (quite often) people's individual machines and they join and leave the P2P system at will, creating considerable hardship in the management of data.

In this chapter, we focus on this modern incarnation of P2P systems. In these systems, the following requirements are typically cited [Daswani et al., 2003]:

- **Autonomy.** An autonomous peer should be able to join or leave the system at any time without restriction. It should also be able to control the data it stores and which other peers can store its data (e.g., some other trusted peers).
- **Query expressiveness.** The query language should allow the user to describe the desired data at the appropriate level of detail. The simplest form of query

is key look-up, which is only appropriate for finding files. Keyword search with ranking of results is appropriate for searching documents, but for more structured data, an SQL-like query language is necessary.

- **Efficiency.** The efficient use of the P2P system resources (bandwidth, computing power, storage) should result in lower cost, and, thus, higher throughput of queries, i.e., a higher number of queries can be processed by the P2P system in a given time interval.
- **Quality of service.** This refers to the user-perceived efficiency of the system, such as completeness of query results, data consistency, data availability, query response time, etc.
- **Fault-tolerance.** Efficiency and quality of service should be maintained despite the failures of peers. Given the dynamic nature of peers that may leave or fail at any time, it is important to properly exploit data replication.
- **Security.** The open nature of a P2P system gives rise to serious security challenges since one cannot rely on trusted servers. With respect to data management, the main security issue is access control which includes enforcing intellectual property rights on data contents.

A number of different uses of P2P systems have been developed [Valduriez and Pacitti, 2004]: they have been successfully used for sharing computation (e.g., SETI@home – http://www.setiathome.ssl.berkeley.edu), communication (e.g., ICQ – http://www.icq.com), or data sharing (e.g., Gnutella – http://www.gnutelliums.com – and Kazaa – http://www.kazaa.com). Our interest, naturally, is on data sharing systems. The commercial systems (such as Gnutella, Kazaa and others) are quite limited when viewed from the perspective of database functionality. Two important limitations are that they provide only file level sharing with no sophisticated content-based search/query facilities, and they are single-application systems that focus on performing one task, and it is not straightforward to extend them for other applications/functions [Ooi et al., 2003b]. In this chapter, we discuss the research activities towards providing proper database functionality over P2P infrastructures. Within this context, data management issues that must be addressed include the following:

- Data location: peers must be able to refer to and locate data stored in other peers.
- Query processing: given a query, the system must be able to discover the peers that contribute relevant data and efficiently execute the query.
- Data integration: when shared data sources in the system follow different schemas or representations, peers should still be able to access that data, ideally using the data representation used to model their own data.
- Data consistency: if data are replicated or cached in the system, a key issue is to maintain the consistency between these duplicates.

Figure 16.1 shows a reference architecture for a peer participating in a data sharing P2P system. Depending on the functionality of the P2P system, one or more

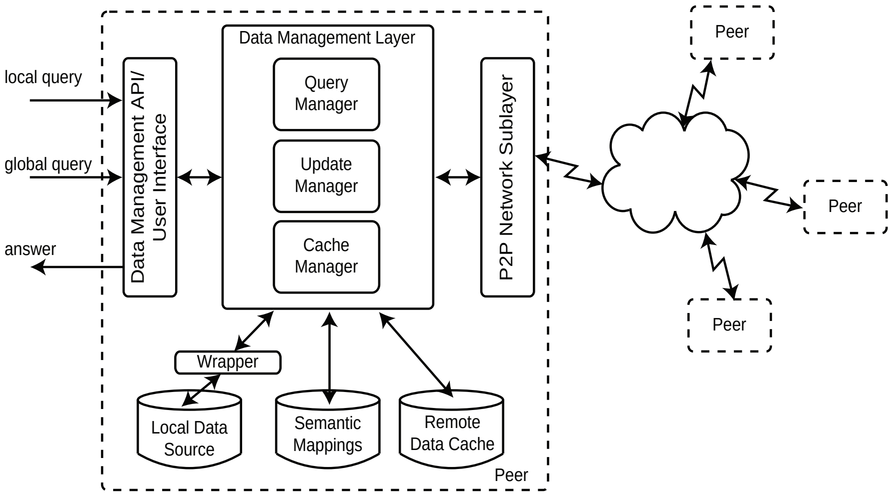


Fig. 16.1 Peer Reference Architecture

of the components in the reference architecture may not exist, may be combined together, or may be implemented by specialized peers. The key aspect of the proposed architecture is the separation of the functionality into three main components: (1) an interface used for submitting the queries; (2) a data management layer that handles query processing and metadata information (e.g., catalogue services); and (3) a P2P infrastructure, which is composed of the P2P network sublayer and P2P network. In this chapter, we focus on the P2P data management layer and P2P infrastructure.

Queries are submitted using a user interface or data management API and handled by the data management layer. Queries may refer to data stored locally or globally in the system. The query request is processed by a query manager module that retrieves semantic mapping information from a repository when the system integrates heterogeneous data sources. This semantic mapping repository contains meta-information that allows the query manager to identify peers in the system with data relevant to the query and to reformulate the original query in terms that other peers can understand. Some P2P systems may store the semantic mapping in specialized peers. In this case, the query manager will need to contact these specialized peers or transmit the query to them for execution. If all data sources in the system follow the same schema, neither the semantic mapping repository nor its associated query reformulation functionality are required.

Assuming a semantic mapping repository, the query manager invokes services implemented by the P2P network sublayer to communicate with the peers that will be involved in the execution of the query. The actual execution of the query is influenced by the implementation of the P2P infrastructure. In some systems, data are sent to the peer where the query was initiated and then combined at this peer. Other systems provide specialized peers for query execution and coordination. In either case, result data returned by the peers involved in the execution of the query may be cached locally to speed up future executions of similar queries. The cache manager maintains the local cache of each peer. Alternatively, caching may occur only at specialized peers.

The query manager is also responsible for executing the local portion of a global query when data are requested by a remote peer. A wrapper may hide data, query language, or any other incompatibilities between the local data source and the data management layer. When data are updated, the update manager coordinates the execution of the update between the peers storing replicas of the data being updated.

The P2P network infrastructure, which can be implemented as either structured or unstructured network topology, provides communication services to the data management layer.

In the remainder of this chapter, we will address each component of this reference architecture, starting with infrastructure issues in Section 16.1. The problems of data mapping and the approaches to address them are the topics of Section 16.2. Query processing is discussed in Section 16.3. Data consistency and replication issues are discussed in Section 16.4.

16.1 Infrastructure

The infrastructure of all P2P systems is a P2P network, which is built on top of a physical network (usually the Internet); thus it is commonly referred to as the *overlay network*. The overlay network may (and usually does) have a different topology than the physical network and all the algorithms focus on optimizing communication over the overlay network (usually in terms of minimizing the number of "hops" that a message needs to go through from a source node to a destination node – both in the overlay network). The possible disconnect between the overlay network and the physical network may be a problem in that two nodes that are neighbors in the overlay network may, in some cases, be considerably far apart in the physical network. Therefore, the cost of communication within the overlay network may not reflect the actual cost of communication in the physical network. We address this issue at the appropriate points during the infrastructure discussion.

Overlay networks can be of two general types: pure and hybrid. *Pure overlay networks* (more commonly referred to as *pure P2P networks*) are those where there is no differentiation between any of the network nodes – they are all equal. In *hybrid P2P networks*, on the other hand, some nodes are given special tasks to perform. Hybrid networks are commonly known as *super-peer systems*, since some of the peers are responsible for "controlling" a set of other peers in their domain. The pure networks can be further divided into structured and unstructured networks. *Structured networks* tightly control the topology and message routing, whereas in *unstructured networks* each node can directly communicate with its neighbors and can join the network by attaching themselves to any node.

16.1.1 Unstructured P2P Networks

Unstructured P2P networks refer to those with no restriction on data placement in the overlay topology. The overlay network is created in a nondeterministic (ad hoc) manner and the data placement is completely unrelated to the overlay topology. Each peer knows its neighbors, but does not know the resources that they have. Figure 16.2 shows an example unstructured P2P network.

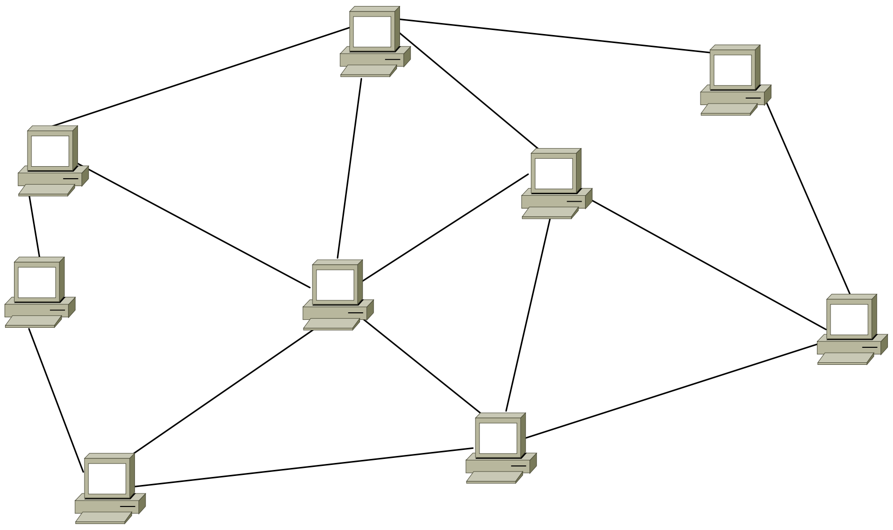


Fig. 16.2 Unstructured P2P Network

Unstructured networks are the earliest examples of P2P systems whose core functionality was (and remains) file sharing. In these systems replicated copies of popular files are shared among peers, without the need to download them from a centralized server. Examples of these systems are Napster (http://www.napster.com), Gnutella, Freenet [Clarke et al., 2000, 2002], Kazaa, and BitTorrent (http://www.bittorrent.com).

A fundamental issue in all P2P networks is the type of index to the resources that each peer holds, since this determines how resources are searched. Note that what is called "index management" in the context of P2P systems is very similar to catalog management that we studied in Chapter 3. Indexes are stored metadata that the system maintains. The exact content of the metadata differs in different P2P systems. In general, it includes, at a minimum, information on the resources and sizes.

There are two alternatives to maintaining indices: centralized, where one peer stores the metadata for the entire P2P system, and distributed, where each peer maintains metadata for resources that it holds. Again, the alternatives are identical to those for directory management. Napster is an example of a system that maintains a centralized index, while Gnutella maintains a distributed one.

The type of index supported by a P2P system (centralized or distributed) impacts how resources are searched. Note that we are not, at this point, referring to running queries; we are merely discussing how, given a resource identifier, the underlying P2P infrastructure can locate the relevant resource. In systems that maintain a centralized index, the process involves consulting the central peer to find the location of the resource, followed by directly contacting the peer where the resource is located (Figure 16.3). Thus, the system operates similar to a client/server one up to the point of obtaining the necessary index information (i.e., the metadata), but from that point on, the communication is only between the two peers. Note that the central peer may return a set of peers who hold the resource and the requesting peer may choose one among them, or the central peer may make the choice (taking into account loads and network conditions, perhaps) and return only a single recommended peer.

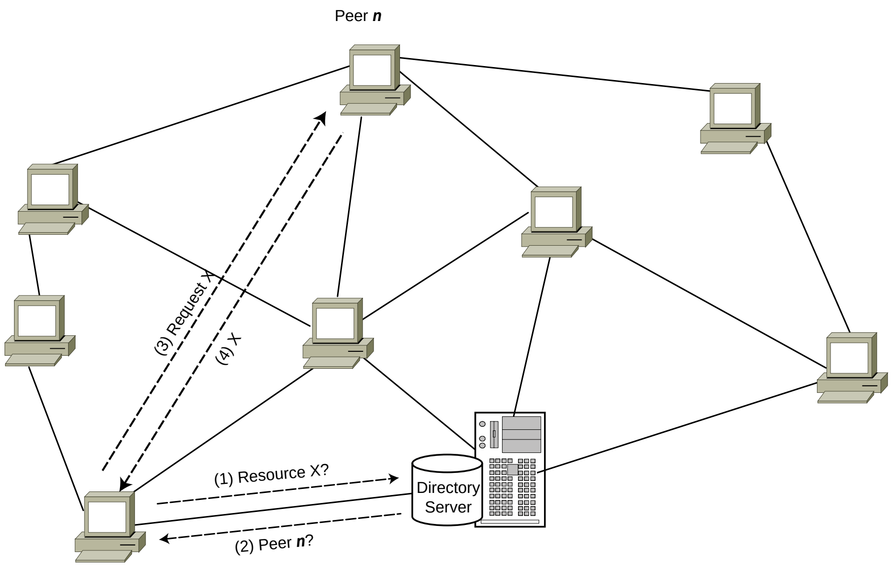


Fig. 16.3 Search over a Centralized Index. (1) A peer asks the central index manager for resource, (2) The response identifies the peer with the resource, (3) The peer is asked for the resource, (4) It is transferred.

In systems that maintain a distributed index, there are a number of search alternatives. The most popular one is flooding, where the peer looking for a resource sends the search request to all of its neighbors on the overlay network. If any of these neighbors have the resource, they respond; otherwise, each of them forwards the request to its neighbors until the resource is found or the overlay network is fully spanned (Figure 16.4).

Naturally, flooding puts very heavy demands on network resources and is not scalable – as the overlay network gets larger, more communication is initiated. This has been addressed by establishing a Time-to-Live (TTL) limit that restricts the

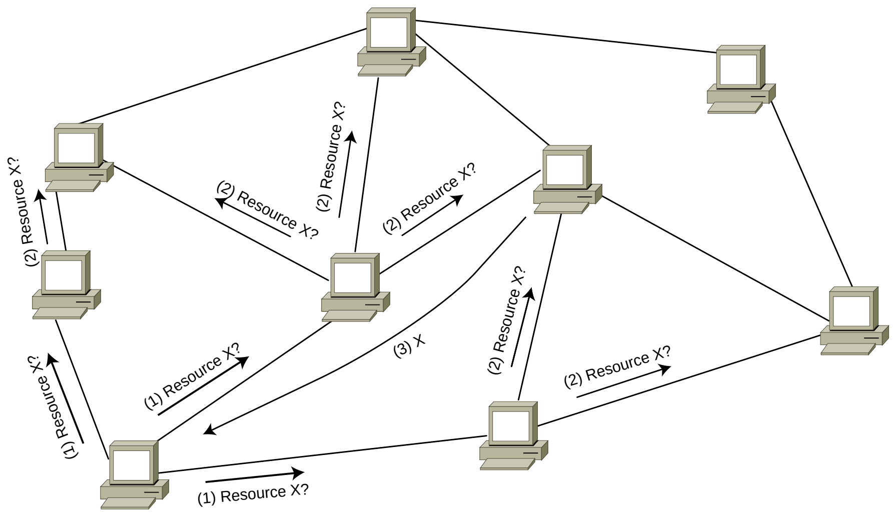


Fig. 16.4 Search over a Decentralized Index. (1) A peer sends the request for resource to all its neighbors, (2) Each neighbor propagates to its neighbors if it doesn't have the resource, (3) The peer who has the resource responds by sending the resource.

number of hops that a request message makes before it is dropped from the network. However, TTL also restricts the number of nodes that are reachable.

There have been other approaches to address this problem. A straightforward method is for each peer to choose a subset of its neighbors and forward the request only to those [Kalogeraki et al., 2002]. How this subset can be determined may vary. For example, the concept of random walks can be used [Lv et al., 2002] where each peer chooses a neighbor at random and propagates the request only to it. Alternatively, each neighbor can maintain not only indices for local resources, but also for resources that are on peers within a radius of itself and use the historical information about their performance in routing queries [Yang and Garcia-Molina, 2002]. Still another alternative is to use similar indices based on resources at each node to provide a list of neighbors that are most likely to be in the direction of the peer holding the requested resources [Crespo and Garcia-Molina, 2002]. These are referred to as routing indices and are used more commonly in structured networks, where we discuss them in more detail.

Another approach is to exploit *gossip protocols*, also known as *epidemic protocols* [Kermarrec and van Steen, 2007]. Gossiping has been initially proposed to maintain the mutual consistency of replicated data by spreading replica updates to all nodes over the network [Demers et al., 1987]. It has since been successfully used in P2P networks for data dissemination. Basic gossiping is simple. Each node in the network has a complete view of the network (i.e., a list of all nodes' addresses) and chooses a node at random to spread the request. The main advantage of gossiping is robustness over node failures since, with very high probability, the request is eventually propagated to all the nodes in the network. In large P2P networks, however, the basic gossiping model does not scale as maintaining the complete view of the network at each node would generate very heavy communication traffic. A solution to scalable gossiping is to maintain at each node only a partial view of the network, e.g., a list of tens of neighbour nodes [Voulgaris et al., 2003]. To gossip a request, a node chooses, at random, a node in its partial view and sends it the request. In addition, the nodes involved in a gossip exchange their partial views to reflect network changes in their own views. Thus, by continuously refreshing their partial views, nodes can self-organize into randomized overlays that scale up very well.

The final issue that we would like to discuss with respect to unstructured networks is how peers join and leave the network. The process is different for centralized versus distributed index approaches. In a centralized index system, a peer that wishes to join simply notifies the central index peer and informs it of the resources that it wishes to contribute to the P2P system. In the case of a distributed index, the joining peer needs to know one other peer in the system to which it "attaches" itself by notifying it and receiving information about its neighbors. At that point, the peer is part of the system and starts building its own neighbors. Peers that leave the system do not need to take any special action, they simply disappear. Their disappearance will be detected in time, and the overlay network will adjust itself.

16.1.2 Structured P2P Networks

Structured P2P networks have emerged to address the scalability issues faced by unstructured P2P networks [Ritter, 2001; Ratnasamy et al., 2001b; Stoica et al., 2001a]. They achieve this goal by tightly controlling the overlay topology and the placement of resources. Thus, they achieve higher scalability at the expense of lower autonomy as each peer that joins the network allows its resources to be placed on the network based on the particular control method that is used.

As with unstructured P2P networks, there are two fundamental issues to be addressed: how are the resources indexed, and how are they searched. The most popular indexing and data location mechanism that is used in structured P2P networks is *dynamic hash table* (DHT). DHT-based systems provide two API's: `put(key, data)` and `get(key)`, where key is an object identifier. The key is hashed to generate a peer id, which stores the data corresponding to object contents (Figure 16.5). Dynamic hashing has also been successfully used to address the scalability issues of very large distributed file structures [Devine, 1993; Litwin et al., 1993].

A straightforward approach could be to use the URI of the resource as the IP address of the peer that would hold the resource [Harvey et al., 2003]. However, one of the important design requirements is to provide a uniform distribution of resources over the overlay network and URIs/IP addresses do not provide sufficient flexibility. Consequently, *consistent hashing* techniques that provide uniform hashing of values are used to evenly place the data on the overlay. Although many hash functions may be employed for generating *virtual address mappings* for the resource, SHA-1 has

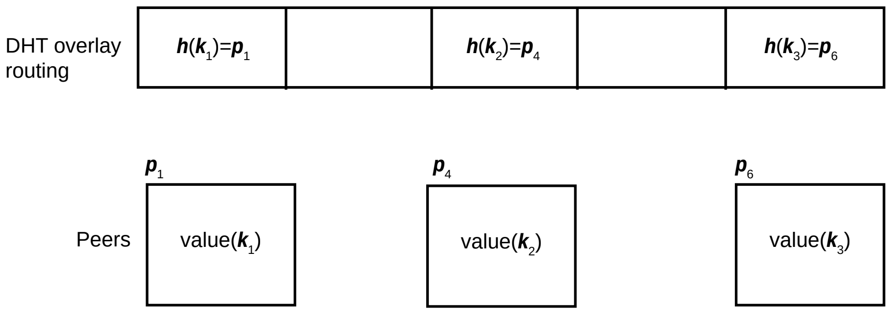


Fig. 16.5 DHT-based P2P Network

become the most widely accepted *base*[1] hash function that supports both uniformity as well as security (by supporting data-integrity for the keys). The actual design of the hash function may be implementation dependent and we won't discuss that issue any further.

Search (commonly called "lookup") over a DHT-based structured P2P network also involves the hash function: the key of the resource is hashed to get the id of the peer in the overlay network that is responsible for that key. The lookup is then initiated on the overlay network to locate the target node in question. This is referred to as the *routing protocol*, and it differs between different implementations and is closely associated with the overlay structure used. We will discuss one example approach shortly.

While all routing protocols aim to provide efficient lookups, they also try to minimize the *routing information* (also called *routing state*) that needs to be maintained in a routing table at each peer in the overlay. This information differs between various routing protocols and overlay structures, but it needs to provide sufficient directory-type information to route the `put` and `get` requests to the appropriate peer on the overlay. All routing table implementations require the use of maintenance algorithms in order to keep the routing state up-to-date and consistent. In contrast to routers on the Internet that also maintain routing databases, P2P systems pose a greater challenge since they are characterized by high node volatility and undependable network links. Since DHTs also need to support perfect recall (i.e., all the resources that are accessible through a given key have to be found), routing state consistency becomes a key challenge. Therefore, the maintenance of consistent routing state in the face of concurrent lookups and during periods of high network volatility is essential.

Many DHT-based overlays have been proposed. These can be categorized according to their *routing geometry* and *routing algorithm* [Gummadi et al., 2003]. Routing geometry essentially defines the manner in which neighbors and routes are arranged. The routing algorithm corresponds to the routing protocol discussed above

[1] A base hash function is defined as a function that is used as a basis for the design of another hash function.

and is defined as the manner in which next-hops/routes are chosen on a given routing geometry. The more important existing DHT-based overlays can be categorized as follows:

- **Tree.** In the tree approach, the leaf nodes correspond to the node identifiers that store the keys to be searched. The height of the tree is $\log(n)$, where n is the number of nodes in the tree. The search proceeds from the root to the leaves by doing a longest prefix match at each of the intermediate nodes until the target node is found. Therefore, in this case, matching can be thought of as correcting bit values from left-to-right at each successive hop in the tree. A popular DHT implementation that falls into this category is Tapestry [Zhao et al., 2004], which uses *surrogate routing* in order to forward requests at each node to the closest digit in the routing table. Surrogate routing is defined as routing to the *closest* digit when an exact match in the longest prefix cannot be found. In Tapestry, each unique identifier is associated with a node that is the root of a unique spanning tree used to route messages for the given identifier. Therefore, lookups proceed from the base of the spanning tree all the way to the root node of the identifier. Although this is somewhat different from traditional tree structures, Tapestry routing geometry is very closely associated to a tree structure and we classify it as such.
  In tree structures, a node in the system has 2^{i-1} nodes to choose from as its neighbor from the subtree with whom it has $\log(n-i)$ prefix bits in common. The number of potential neighbors increases exponentially as we proceed further up in the tree. Thus, in total there are $n^{\log(n)/2}$ possible routing tables per node (note, however that, only one such routing table can be selected for a node). Therefore, the tree geometry has good neighbor selection characteristics that would provide it with fault tolerance. However, routing can only be done through one neighboring node when sending to a particular destination. Consequently, the tree-structured DHTs do not provide any flexibility in the selection of routes.
- **Hypercube.** The hypercube routing geometry is based on d-dimensional Cartesian coordinate space that is partitioned into an individual set of zones such that each node maintains a separate zone of the coordinate space. An example of hypercube-based DHT is the Content Addressable Network (CAN) [Ratnasamy et al., 2001a]. The number of neighbors that a node may have in a d-dimensional coordinate space is $2d$ (for the sake of discussion, we consider $d = \log(n)$). If we consider each coordinate to represent a set of bits, then each node identifier can be represented as a bit string of length $\log(n)$. In this way, the hypercube geometry is very similar to the tree since it also simply *fixes* the bits at each hop to reach the destination. However, in the hypercube, since the bits of neighboring nodes only differ in *exactly* one bit, each forwarding node needs to modify only a single bit in the bit string, which can be done in any order. Thus, if we consider the correction of the bit string, the first correction can be applied to any $\log(n)$ nodes, the next correction can be applied to any $\log(n)-1$ nodes, etc. Therefore, we have $\log(n)!$ possible routes between

nodes which provides high route flexibility in the hypercube routing geometry. However, a node in the coordinate space does not have any choice over its neighbors' coordinates since adjacent coordinate zones in the coordinate space can't change. Therefore, hypercubes have poor neighbor selection flexibility.

- **Ring.** The ring geometry is represented as a one-dimensional circular identifier space where the nodes are placed at different locations on the circle. The distance between any two nodes on the circle is the numeric identifier difference (clockwise) around the circle. Since the circle is one-dimensional, the data identifiers can be represented as single decimal digits (represented as binary bit strings) that map to a node that is closest in the identifier space to the given decimal digit. Chord [Stoica et al., 2001b] is a popular example of the ring geometry. Specifically, in Chord, a node whose identifier is a maintains information about $\log(n)$ other neighbors on the ring where the i^{th} neighbor is the node closest to $a+2^{i-1}$ on the circle. Using these links (called *fingers*), Chord is able to route to any other node in $\log(n)$ hops.
  A careful analysis of Chord's structure reveals that a node does not necessarily need to maintain the node closest to $a+2^{i-1}$ as its neighbor. In fact, it can still maintain the $\log(n)$ lookup upper bound if any node from the range $[(a+2^{i-1}),(a+2^{i})]$ is chosen. Therefore, in terms of route flexibility, it is able to select between $n^{\log(n)/2}$ routing tables for each node. This provides a great deal of neighbor selection flexibility. Moreover, for routing to any node, the first hop has $\log(n)$ neighbors that can route the search to the destination and the next node has $\log(n)-1$ nodes, and so on. Therefore, there are typically $\log(n)!$ possible routes to the destination. Consequently, ring geometry also provides good route selection flexibility.

In addition to these most popular geometries, there have been many other DHT-based structured overlays that have been proposed that use different topologies. Some of these are Viceroy [Malkhi et al., 2002], Kademlia [Maymounkov and Mazières, 2002], and Pastry [Rowstron and Druschel, 2001].

DHT-based overlays are efficient in that they guarantee finding the node on which to place or find the data in $\log(n)$ hops where n is the number of nodes in the system. However, they have a number of problems, in particular when viewed from the data management perspective. One of the issues with DHTs that employ consistent hashing functions for better distribution of resources is that two peers that are "neighbors" in the overlay network because of the proximity of their hash values may be geographically quite apart in the actual network. Thus, communicating with a neighbor in the overlay network may incur high transmission delays in the actual network. There have been studies to overcome this difficulty by designing *proximity-aware* or *locality-aware* hash functions. Another difficulty is that they do not provide any flexibility in the placement of data – a data item has to be placed on the node that is determined by the hash function. Thus, if there are P2P nodes that contribute their own data, they need to be willing to have data moved to other nodes. This is problematic from the perspective of node autonomy. The third difficulty is in that it is hard to run range queries over DHT-based architectures since, as is well-known, it is hard to run range queries over hash indices. There have been studies to overcome this difficulty that we discuss later.

These concerns have caused the development of structured overlays that do not use DHT for routing. In these systems, peers are mapped into the data space rather than the hash key space. There are multiple ways to partition the data space among multiple peers.

- **Hierarchical structure.** Many systems employ hierarchical overlay structures, including trie, balanced trees, randomized balance trees (e.g., skip list [Pugh, 1989]), and others. Specifically PHT [Ramabhadran et al., 2004] and P-Grid [Aberer, 2001; Aberer et al., 2003a] employ a binary trie structure, where peers whose data share common prefixes cluster under common branches. Balanced trees are also widely used due to their guaranteed routing efficiency (the expected "hop length" between arbitrary peers is proportional to the tree height). For instance, BATON [Jagadish et al., 2005], VBI-tree [Jagadish et al., 2005], and BATON* [Jagadish et al., 2006] employ k-way balanced tree structure to manage peers, and data are evenly partitioned among peers at the leaf-level. In comparison, P-Tree [Crainiceanu et al., 2004] uses a B-tree structure with better flexibility on tree structural changes. SkipNet [Harvey et al., 2003] and Skip Graph [Aspnes and Shah, 2003] are based on the skip list, and they link peers according to a randomized balanced tree structure where the node order is determined by each node's data values.
- **Space-filling curve.** This architecture is usually used to linearize sort data in multi-dimensional data space. Peers are arranged along the space-filling curve (e.g., Hilbert curve) so that sorted traversal of peers according to data order is possible [Schmidt and Parashar, 2004].
- **Hyper-rectangle structure.** In these systems, each dimension of the hyper-rectangle corresponds to one attribute of the data according to which an organization is desired. Peers are distributed in the data space either uniformly or based on data locality (e.g., through data intersection relationship). The hyper-rectangle space is then partitioned by peers based on their geometric positions in the space, and neighboring peers are interconnected to form the overlay network [Ganesan et al., 2004].

16.1.3 Super-peer P2P Networks

Super-peer P2P systems are hybrid between pure P2P systems and the traditional client-server architectures. They are similar to client-server architectures in that not all peers are equal; some peers (called *super-peers*) act as dedicated serves for some other peers and can perform complex functions such as indexing, query processing, access control, and meta-data management. If there is only one super-peer in the system, then this reduces to the client-server architecture. They are considered P2P systems, however, since the organization of the super-peers follow P2P organization, and super-peers can communicate with each other in sophisticated ways. Thus, unlike client-server systems, global information is not necessarily centralized and can be partitioned or replicated across super-peers.

In a super-peer network, a requesting peer sends the request, which can be expressed in a high-level language, to its responsible super-peer. The super-peer can then find the relevant peers either directly through its index or indirectly using its neighbor super-peers. More precisely, the search for a resource proceeds as follows (see Figure 16.6):

1. A peer, say Peer 1, asks for a resource by sending a request to its super-peer.
2. If the resource exists at one of the peers controlled by this super-peer, it notifies Peer 1, and the two peers then communicate to retrieve the resource. Otherwise, the super-peer sends the request to the other super-peers.
3. If the resource does not exist at one of the peers controlled by this super-peer, the super-peer asks the other super-peers. The super-peer of the node that contains the resource (say Peer *n*) responds to the requesting super-peer.
4. Peer *n*'s identity is sent to Peer 1, after which the two peers can communicate directly to retrieve the resource.

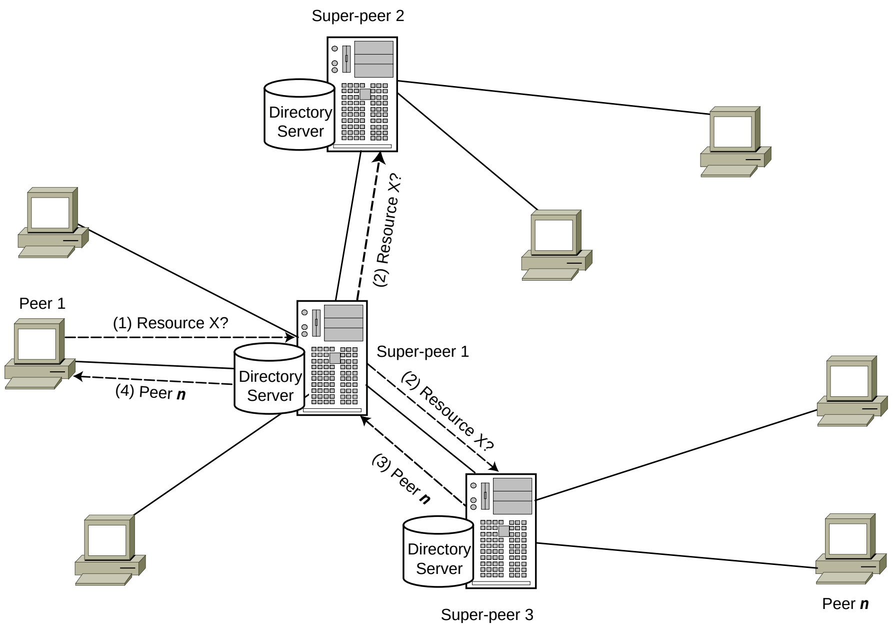


Fig. 16.6 Search over a Super-peer System. (1) A peer sends the request for resource to all its super-peer, (2) The super-peer sends the request to other super-peers if necessary, (3) The super-peer one of whose peers has the resource responds by indicating that peer, (4) The super-peer notifies the original peer.

REQUIREMENTS	UNSTRUCTURED	STRUCTURED	SUPER-PE
Autonomy	Low	Low	Moderate
Query expressiveness	High	Low	High
Efficiency	Low	High	High
QoS	Low	High	High
Fault-tolerance	High	High	Low
Security	Low	Low	High

Fig. 16.7 Comparison of Approaches.

The main advantages of super-peer networks are efficiency and quality of service (e.g., completeness of query results, query response time, etc.). The time needed to find data by directly accessing indices in a super-peer is very small compared with flooding. In addition, super-peer networks exploit and take advantage of peers' different capabilities in terms of CPU power, bandwidth, or storage capacity as super-peers take on a large portion of the entire network load. Access control can also be better enforced since directory and security information can be maintained at the super-peers. However, autonomy is restricted since peers cannot log in freely to any super-peer. Fault-tolerance is typically lower since super-peers are single points of failure for their sub-peers (dynamic replacement of super-peers can alleviate this problem).

Examples of super-peer networks include Edutella [Nejdl et al., 2003] and JXTA (http://www.jxta.org).

16.1.4 Comparison of P2P Networks

Figure 16.7 summarizes how the requirements for data management (autonomy, query expressiveness, efficiency, quality of service, fault-tolerance, and security) are possibly attained by the three main classes of P2P networks. This is a rough comparison to understand the respective merits of each class. Obviously, there is room for improvement in each class of P2P networks. For instance, fault-tolerance can be improved in super-peer systems by relying on replication and fail-over techniques. Query expressiveness can be improved by supporting more complex queries on top of structured networks.

16.2 Schema Mapping in P2P Systems

We discussed the importance of, and the techniques for, designing database integration systems in Chapter 4. Similar issues arise in data sharing P2P systems.

Due to specific characteristics of P2P systems, e.g., the dynamic and autonomous nature of peers, the approaches that rely on centralized global schemas no longer apply. The main problem is to support decentralized schema mapping so that a query expressed on one peer's schema can be reformulated to a query on another peer's schema. The approaches which are used by P2P systems for defining and creating the mappings between peers' schemas can be classified as follows: pairwise schema mapping, mapping based on machine learning techniques, common agreement mapping, and schema mapping using information retrieval (IR) techniques.

16.2.1 Pairwise Schema Mapping

In this approach, each user defines the mapping between the local schema and the schema of any other peer that contains data that are of interest. Relying on the transitivity of the defined mappings, the system tries to extract mappings between schemas that have no defined mapping.

Piazza [Tatarinov et al., 2003] follows this approach (see Figure 16.8). The data are shared as XML documents, and each peer has a schema that defines the terminology and the structural constraints of the peer. When a new peer (with a new schema) joins the system for the first time, it maps its schema to the schema of some other peers in the system. Each mapping definition begins with an XML template that matches some path or subtree of an instance of the target schema. Elements in the template may be annotated with query expressions that bind variables to XML nodes in the source. Active XML [Abiteboul et al., 2002, 2008b] also relies on XML documents for data sharing. The main innovation is that XML documents are active in the sense that they can include Web service calls. Therefore, data and queries can be seamlessly integrated. We discuss this further in Chapter 17.

The Local Relational Model (LRM) [Bernstein et al., 2002] is another example that follows this approach. LRM assumes that the peers hold relational databases, and each peer knows a set of peers with which it can exchange data and services. This set of peers is called peer's *acquaintances*. Each peer must define semantic dependencies and translation rules between its data and the data shared by each of its acquaintances. The defined mappings form a semantic network, which is used for query reformulation in the P2P system. Hyperion [Kementsietsidis et al., 2003] generalizes this approach to deal with autonomous peers that form acquaintances at run-time, using mapping tables to define value correspondences among heterogeneous databases. Peers perform local querying and update processing, and also propagate queries and updates to their acquainted peers.

PGrid [Aberer et al., 2003b] also assumes the existence of pairwise mappings between peers, initially constructed by skilled experts. Relying on the transitivity of these mappings and using a gossip algorithm, PGrid extracts new mappings that relate the schemas of the peers between which there is no predefined schema mapping.

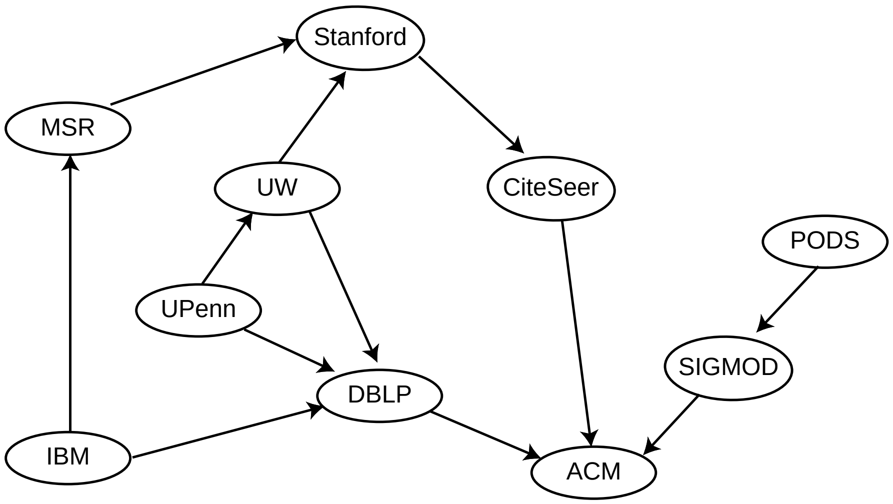


Fig. 16.8 An Example of Pairwise Schema Mapping in Piazza

16.2.2 Mapping based on Machine Learning Techniques

This approach is generally used when the shared data are defined based on ontologies and taxonomies as proposed for the semantic web. It uses machine learning techniques to automatically extract the mappings between the shared schemas. The extracted mappings are stored over the network, in order to be used for processing future queries. GLUE [Doan et al., 2003b] uses this approach. Given two ontologies, for each concept in one, GLUE finds the most similar concept in the other. It gives well founded probabilistic definitions to several practical similarity measures, and uses multiple learning strategies, each of which exploits a different type of information either in the data instances or in the taxonomic structure of the ontologies. To further improve mapping accuracy, GLUE incorporates commonsense knowledge and domain constraints into the schema mapping process. The basic idea is to provide classifiers for the concepts. To decide the similarity between two concepts A and B, the data of concept B are classified using A's classifier and vice versa. The amount of values that can be successfully classified into A and B represent the similarity between A and B.

16.2.3 Common Agreement Mapping

In this approach, the peers that have a common interest agree on a common schema description for data sharing. The common schema is usually prepared and maintained by expert users. APPA [Akbarinia et al., 2006a; Akbarinia and Martins, 2007] makes the assumption that peers wishing to cooperate, e.g., for the duration of an experiment, agree on a Common Schema Description (CSD). Given a CSD, a peer schema can be specified using views. This is similar to the LAV approach in data integration systems, except that queries at a peer are expressed in terms of the local views, not the CSD. Another difference between this approach and LAV is that the CSD is not a global schema, i.e., it is common to a limited set of peers with a common interest (see Figure 16.9). Thus, the CSD does not pose scalability challenges. When a peer decides to share data, it needs to map its local schema to the CSD.

Example 16.1. Given two CSD relation definitions r_1 and r_2, an example of peer mapping at peer p is:

$$p : r(A,B,D) \subseteq csd : r_1(A,B,C), csd : r_2(C,D,E)$$

In this example, the relation $r(A,B,D)$ that is shared by peer p is mapped to relations $r_1(A,B,C)$, $r_2(C,D,E)$ both of which are involved in the CSD. In APPA, the mappings between the CSD and each peer's local schema are stored locally at the peer. Given a query Q on the local schema, the peer reformulates Q to a query on the CSD using locally stored mappings. ♦

AutoMed [McBrien and Poulovassilis, 2003] is another system that relies on common agreements for schema mapping. It defines the mappings by using primitive bidirectional transformations defined in terms of a low-level data model.

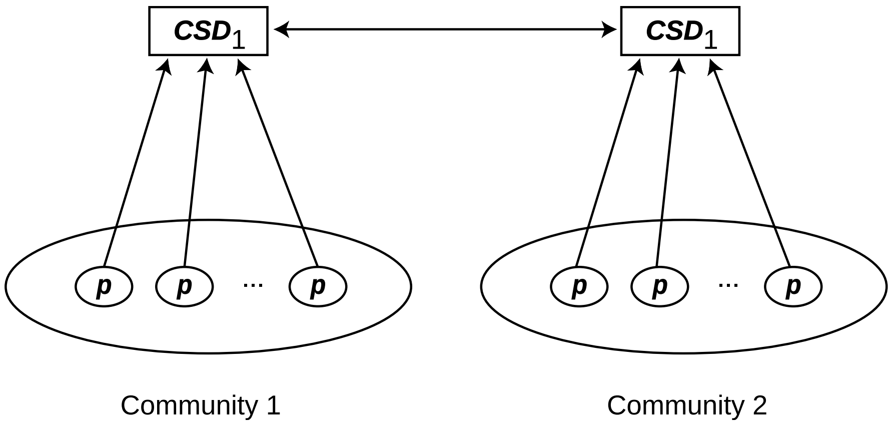


Fig. 16.9 Common Agreement Schema Mapping in APPA

16.2.4 Schema Mapping using IR Techniques

This approach extracts the schema mappings at query execution time using IR techniques by exploring the schema descriptions provided by users. PeerDB [Ooi et al., 2003a] follows this approach for query processing in unstructured P2P networks. For each relation that is shared by a peer, the description of the relation and its attributes is maintained at that peer. The descriptions are provided by users upon creation of relations, and serve as a kind of synonymous names of relation names and attributes. When a query is issued, a request to find out potential matches is produced and flooded to the peers that return the corresponding metadata. By matching keywords from the metadata of the relations, PeerDB is able to find relations that are potentially similar to the query relations. The relations that are found are presented to the issuer of the query who decides whether or not to proceed with the execution of the query at the remote peer that owns the relations.

Edutella [Nejdl et al., 2003] also follows this approach for schema mapping in super-peer networks. Resources in Edutella are described using the RDF metadata model, and the descriptions are stored at super-peers. When a user issues a query at a peer *p*, the query is sent to *p*'s super-peer where the stored schema descriptions are explored and the addresses of the relevant peers are returned to the user. If the super-peer does not find relevant peers, it sends the query to other super-peers such that they search relevant peers by exploring their stored schema descriptions. In order to explore stored schemas, super-peers use the RDF-QEL query language, which is based on Datalog semantics and thus compatible with all existing query languages, supporting query functionalities that extend the usual relational query languages.

16.3 Querying Over P2P Systems

P2P networks provide basic techniques for routing queries to relevant peers and this is sufficient for supporting simple, exact-match queries. For instance, as noted earlier, a DHT provides a basic mechanism to efficiently look up data based on a key value. However, supporting more complex queries in P2P systems, particularly in DHTs, is difficult and has been the subject of much recent research. The main types of complex queries which are useful in P2P systems are top-k queries, join queries, and range queries. In this section, we discuss the techniques for processing them.

16.3.1 Top-k Queries

Top-k queries have been used in many domains such as network and system monitoring, information retrieval, and multimedia databases [Ilyas et al., 2008]. With a top-k query, the user requests *k* most relevant answers to be returned by the system. The degree of relevance (score) of the answers to the query is determined by a scoring function. Top-k queries are very useful for data management in P2P systems, in particular when the number of all the answers is very large [Akbarinia et al., 2006b].

Example 16.2. Consider a P2P system with medical doctors who want to share some (restricted) patient data for an epidemiological study. Assume that all doctors agreed on a common Patient description in relational format. Then, one doctor may want to submit the following query to obtain the top 10 answers ranked by a scoring function over height and weight:

```
SELECT      *
FROM        Patient P
WHERE       P.disease = ``diabetes''
AND         P.height < 170
AND         P.weight > 160
ORDER BY    scoring-function(height,weight)
STOP AFTER 10
```

The scoring function specifies how closely each data item matches the conditions. For instance, in the query above, the scoring function could compute the ten most overweight people. ♦

Efficient execution of top-k queries in large-scale P2P systems is difficult. In this section, we first discuss the most efficient techniques proposed for top-k query processing in distributed systems. Then, we present the techniques proposed for P2P systems.

16.3.1.1 Basic Techniques

An efficient algorithm for top-k query processing in centralized and distributed systems is the Threshold Algorithm (TA) [Nepal and Ramakrishna, 1999; Güntzer et al., 2000; Fagin et al., 2003]. TA is applicable for queries where the scoring function is monotonic, i.e., any increase in the value of the input does not decrease the value of the output. Many of the popular aggregation functions such as Min, Max, and Average are monotonic. TA has been the basis for several algorithms, and we discuss these in this section.

Threshold Algorithm (TA).

TA assumes a model based on lists of data items sorted by their local scores [Fagin, 1999]. The model is as follows. Suppose we have m lists of n data items such that each data item has a local score in each list and the lists are sorted according to the local scores of their data items. Furthermore, each data item has an overall score that is computed based on its local scores in all lists using a given scoring function. For example, consider the database (i.e., three sorted lists) in Figure 16.10. Assuming the scoring function computes the sum of the local scores of the same data item in all lists, the overall score of item d_1 is $30+21+14=65$.

Then the problem of top-k query processing is to find the k data items whose overall scores are the highest. This problem model is simple and general. Suppose we want to find the top-k tuples in a relational table according to some scoring function over its attributes. To answer this query, it is sufficient to have a sorted (indexed) list of the values of each attribute involved in the scoring function, and return the k tuples whose overall scores in the lists are the highest. As another example, suppose we want to find the top-k documents whose aggregate rank is the highest with respect to some given set of keywords. To answer this query, the solution is to have, for each keyword, a ranked list of documents, and return the k documents whose aggregate rank over all lists are the highest.

TA considers two modes of access to a sorted list. The first mode is sorted (or sequential) access that accesses each data item in their order of appearance in the list. The second mode is random access by which a given data item in the list is directly looked up, for example, by using an index on item id.

Given the m sorted lists of n data items, TA (see Algorithm 16.1), goes down the sorted lists in parallel, and, for each data item, retrieves its local scores in all lists through random access and computes the overall score. It also maintains in a set Y the k data items whose overall scores are the highest so far. The stopping mechanism of TA uses a threshold that is computed using the last local scores seen under sorted access in the lists. For example, consider the database in Figure 16.10. At position 1 for all lists (i.e., when only the first data items have been seen under sorted access) assuming that the scoring function is the sum of the scores, the threshold is $30+28+30=88$. At position 2, it is 84. Since data items are sorted in the lists in decreasing order of local score, the threshold decreases as one moves down the list. This process continues until k data items are found whose overall scores are greater than a threshold.

Example 16.3. Consider again the database (i.e., three sorted lists) shown in Figure 16.10. Assume a top-3 query Q (i.e., $k=3$), and suppose the scoring function computes the sum of the local scores of the data item in all lists. TA first looks at the data items which are at position 1 in all lists, i.e., d_1, d_2, and d_3. It looks up the local scores of these data items in other lists using random access and computes their overall scores (which are 65, 63 and 70, respectively). However, none of them has an overall score that is as high as the threshold of position 1 (which is 88). Thus, at position 1, TA does not stop. At this position, we have $Y=\{d_1, d_2, d_3\}$, i.e., the k highest scored data items seen so far. At positions 2 and 3, Y is set to $\{d_3, d_4, d_5\}$ and $\{d_3, d_5, d_8\}$ respectively. Before position 6, none of the data items involved in Y has an overall score higher than or equal to the threshold value. At position 6, the threshold value is 63, which is less than the overall score of the three data items involved in Y, i.e., $Y=\{d_3, d_5, d_8\}$. Thus, TA stops. Note that the contents of Y at position 6 is exactly the same as at position 3. In other words, at position 3, Y already contains all top-k answers. In this example, TA does three additional sorted accesses in each list that do not contribute to the final result. This is a characteristic of TA algorithm in that it has a conservative stopping condition that causes it to stop later than necessary – in this example, it performs 9 sorted accesses and $18=(9*2)$ random accesses that do not contribute to the final result. ♦

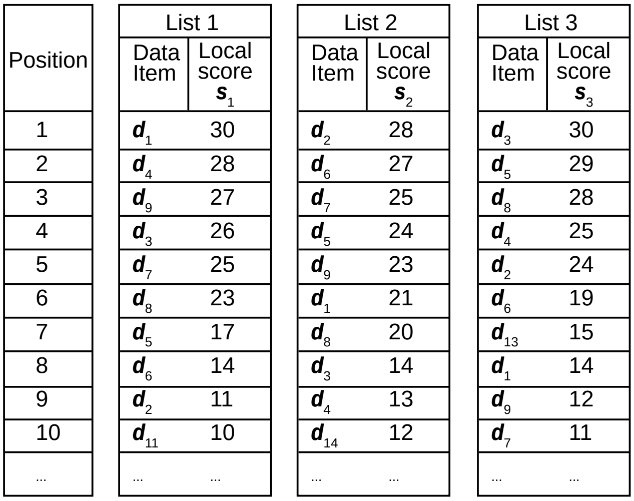
Algorithm 16.1: Threshold Algorithm (TA)

```
Input: $L_1, L_2, \ldots, L_m$: $m$ sorted lists of $n$ data items ;
$f$: scoring function
Output: $Y$: list of top-k data items
begin
    $j \leftarrow 1$ ;
    $threshold \leftarrow 1$ ;
    $min\_overall\_score \leftarrow 0$ ;
    while $j \neq n+1$ and $min\_overall\_score < threshold$ do
        {Do sorted access in parallel to each of the $m$ sorted lists}
        for $i$ from 1 to $m$ in parallel do
            {Process each data item at position $j$}
            for each data item $d$ at position $j$ in $L_i$ do
                {access the local scores of $d$ in the other lists through random
                access}
                $overall\_score(d) \leftarrow f$(scores of $d$ in each $L_i$)
        $Y \leftarrow k$ data items with highest score so far ;
        $min\_overall\_score \leftarrow$ smallest overall score of data items in $Y$ ;
        $threshold \leftarrow f$(local scores at position $j$ in each $L_i$) ;
        $j \leftarrow j+1$
end
```

TA-Style Algorithms.

Several TA-style algorithms, i.e., extensions of TA, have been proposed for distributed top-k query processing. We illustrate these by means of the Three Phase Uniform Threshold (TPUT) algorithm that executes top-k queries in three round trips [Cao and Wang, 2004], assuming that each list is held by one node (which we call the *list holder*) and that the scoring function is sum. The TPUT algorithm (see Algorithm 16.2 executed by the query originator) works as follows.

1. The query originator first gets from each list holder its k top data items. Let f be the scoring function, d be a received data item, and $s_i(d)$ be the local score of d in list L_i. Then the partial sum of d is defined as $psum(d) = \sum_{i=1}^{m} s'_i(d)$ where $s'_i(d) = s_i(d)$ if d has been sent to the coordinator by the holder of L_i, else $s'_i(d) = 0$. The query originator computes the partial sums for all received data items and identifies the items with the k highest partial sums. The partial sum of the $k-$th data item (called *phase-1 bottom*) is denoted by λ_1.
2. The query originator sends a threshold value $\tau = \lambda_1/m$ to every list holder. In response, each list holder sends back all its data items whose local scores are not less than τ. The intuition is that if a data item is not reported by any node in this phase, its score must be less than λ_1, so it cannot be one of the



Position	List 1		List 2		List 3	
	Data Item	Local score s_1	Data Item	Local score s_2	Data Item	Local score s_3
1	d_1	30	d_2	28	d_3	30
2	d_4	28	d_6	27	d_5	29
3	d_9	27	d_7	25	d_8	28
4	d_3	26	d_5	24	d_4	25
5	d_7	25	d_9	23	d_2	24
6	d_8	23	d_1	21	d_6	19
7	d_5	17	d_8	20	d_{13}	15
8	d_6	14	d_3	14	d_1	14
9	d_2	11	d_4	13	d_9	12
10	d_{11}	10	d_{14}	12	d_7	11
...	...	...	...	...	...	...

Fig. 16.10 Example database with 3 sorted lists

top-k data items. Let Y be the set of data items received from list holders. The query originator computes the new partial sums for the data items in Y, and identifies the items with the k highest partial sums. The partial sum of the k-th data item (called phase-2 bottom) is denoted by λ_2. Let the upper bound score of a data item d be defined as $u(d) = \sum_{i=1}^{m} u_i(d)$ where $u_i(d) = s_i(d)$ if d has been received, else $u_i(d) = \tau$. For each data item $d \in D$, if $u(d)$ is less than λ_2, it is removed from Y. The data items that remain in Y are called top-k candidates because there may be some data items in Y that have not been obtained from all list holders. A third phase is necessary to retrieve those.

3. The query originator sends the set of top-k candidate data items to each list holder that returns their scores. Then, it computes the overall score, extracts the k data items with highest scores, and returns the answer to the user.

Example 16.4. Consider the first two sorted lists (List 1 and List 2) in Figure 16.10. Assume a top-2 query Q, i.e., $k = 2$, where the scoring function is sum. Phase 1 produces the sets $Y = \{d_1, d_2, d_4, d_6\}$ and $Z = \{d_1, d_2\}$. Thus we get $\lambda_1/2 = 28/2 = 14$. Let us now denote each data item d in Y as $(d, scoreinList1, scoreinList2)$. Phase 2 produces $Y = \{(d_1,30,21),(d_2,0,28),(d_3,26,14),(d_4,28,0),(d_5,17,24),$ $(d_6,14,27),(d_7,25,25),(d_8,23,20),(d_9,27,23)\}$ and $Z = \{(d_1,30,21),(d_7,25,25)\}$. Note that d_9 could also have been picked instead of d_7 because it has same partial sum. Thus we get λ_2/2=50. The upper bound scores of the data items in Y are obtained as:

$$u(d_1) = 30 + 21 = 51$$
$$u(d_2) = 14 + 28 = 42$$
$$u(d_3) = 26 + 14 = 40$$

Algorithm 16.2: Three Phase Uniform Threshold(TPUT)

```
Input: $L_1, L_2, \ldots, L_m$: $m$ sorted lists of $n$ data items, each at a different list
        holder;
$f$: scoring function
Output: $Y$: list of top-k data items
begin
    {Phase 1}
    for $i$ from 1 to $m$ in parallel do
        $Y \leftarrow$ receive top-k data items from $L_i$ holder
    $Z \leftarrow$ data items with the $k$ highest partial sum in $Y$ ;
    $\lambda_1 \leftarrow$partial sum of $k$-th data item in $Z$ ;
    {Phase 2}
    for $i$ from 1 to $m$ in parallel do
        send $\lambda_1/m$ to $L_i$'s holder ;
        $Y \leftarrow$ all data items from $L_i$'s holder whose local scores are not less than
        $\lambda_1/m$
    $Z \leftarrow$ data items with the $k$ highest partial sum in $Y$ ;
    $\lambda_2 \leftarrow$ partial sum of $k$-th data item in $Z$ ;
    $Y \leftarrow Y - \{$data items in $Y$ whose upper bound score is less than $\lambda_2\}$ ;
    {Phase 3}
    for $i$ from 1 to $m$ in parallel do
        send $Y$ to $L_i$ holder ;
        $Z \leftarrow$ data items from $L_i$'s holder that are in both $Y$ and $L_i$
    $Y \leftarrow k$ data items with highest overall score in $Z$
end
```

$$u(d_4) = 28 + 14 = 42$$
$$u(d_5) = 17 + 24 = 41$$
$$u(d_6) = 14 + 27 = 41$$
$$u(d_7) = 25 + 25 = 50$$
$$u(d_8) = 23 + 20 = 43$$
$$u(d_9) = 27 + 23 = 50$$

After removal of the data items in Y whose upper bound score is less than λ_2, we have $Y = \{d_1, d_7, d_9\}$. The third phase is not necessary in this case as all data items have all their local scores. Thus the final result is $Y = \{d_1, d_7\}$ or $Y = \{d_1, d_9\}$. ♦

When the number of lists (i.e., m) is high, the response time of TPUT is much better than that of the basic TA algorithm [Cao and Wang, 2004].

Best Position Algorithm (BPA).

There are many database instances over which TA keeps scanning the lists although it has seen all top-k answers (as in Example 16.3). Thus, it is possible to stop much sooner. Based on this observation, best position algorithms (BPA) that execute top-k queries much more efficiently than TA have been proposed [Akbarinia et al., 2007a]. The key idea of BPA is that the stopping mechanism takes into account special seen positions in the lists, called the *best positions*. Intuitively, the best position in a list is the highest position such that any position before it has also been seen. The stopping condition is based on the overall score computed using the best positions in all lists.

The basic version of BPA (see Algorithm 16.3) works like TA, except that it keeps track of all positions that are seen under sorted or random access, computes best positions, and has a different stopping condition. For each list L_i, let P_i be the set of positions that are seen under sorted or random access in L_i. Let bp_i, the best position in L_i, be the highest position in P_i such that any position of L_i between 1 and bp_i is also in P_i. In other words, bp_i is best because we are sure that all positions of L_i between 1 and bpi have been seen under sorted or random access. Let $s_i(bp_i)$ be the local score of the data item that is at position bp_i in list L_i. Then, BPA's threshold is $f(s_1(bp_1), s_2(bp_2), \ldots, s_m(bp_m))$ for some function f.

Example 16.5. To illustrate basic BPA, consider again the three sorted lists shown in Figure 16.10 and the query Q in Example 16.3.

1. At position 1, BPA sees the data items d_1, d_2, and d_3. For each seen data item, it does random access and obtains its local score and position in all the lists. Therefore, at this step, the positions that are seen in list L_1 are positions 1, 4, and 9, which are respectively the positions of d_1, d_3 and d_2. Thus, we have $P_1 = \{1,4,9\}$ and the best position in L_1 is $bp_1 = 1$ (since the next position is 4 meaning that positions 2 and 3 have not been seen). For L_2 and L_3 we have $P_2 = \{1,6,8\}$ and $P_3 = \{1,5,8\}$, so $bp_2 = 1$ and $bp_3 = 1$. Therefore, the best positions overall score is $\lambda = f(s_1(1), s_2(1), s_3(1)) = 30+28+30 = 88$. At position 1, the set of the three highest scored data items is $Y = \{d_1, d_2, d_3\}$, and since the overall score of these data items is less than λ , BPA cannot stop.
2. At position 2, BPA sees d_4, d_5, and d_6. Thus, we have $P_1 = \{1,2,4,7,8,9\}$, $P_2 = \{1,2,4,6,8,9\}$ and $P_3 = \{1,2,4,5,6,8\}$. Therefore, we have $bp_1 = 2$, $bp_2 = 2$ and $bp_3 = 2$, so $\lambda = f(s_1(2), s_2(2), s_3(2)) = 28+27+29 = 84$. The overall score of the data items involved in $Y = \{d_3, d_4, d_5\}$ is less than 84, so BPA does not stop.
3. At position 3, BPA sees d_7, d_8, and d_9. Thus, we have $P_1 = P_2 = \{1,2,3,4,5,6,7,8,9\}$, and $P_3 = \{1,2,3,4,5,6,7,8,10\}$. Thus, we have $bp_1 = 9$, $bp_2 = 9$ and $bp_3 = 8$. The best positions overall score is $\lambda = f(s_1(9), s_2(9), s_3(8)) = 11+13+14 = 38$. At this position, we have $Y = \{d_3, d_5, d_8\}$. Since the score of all data items involved in Y is higher than λ, BPA stops, i.e., exactly at the first position where BPA has all top-k answers.

Algorithm 16.3: Best Position Algorithm (BPA)

```
Input: $L_1, L_2, \dots, L_m$: $m$ sorted lists of $n$ data items ;
$f$: scoring function
Output: $Y$: list of top-k data items
begin
    $j \leftarrow 1$ ;
    $threshold \leftarrow 1$ ;
    $min\_overall\_score \leftarrow 0$ ;
    for $i$ from 1 to $m$ in parallel do
        $P_i \leftarrow \emptyset$
    while $j \neq n+1$ and $min\_overall\_score < threshold$ do
        {Do sorted access in parallel to each of the $m$ sorted lists}
        for $i$ from 1 to $m$ in parallel do
            {Process each data item at position $j$}
            for each data item $d$ at position $j$ in $L_i$ do
                {access the local scores of $d$ in the other lists through random
                access}
                $overall\_score(d) \leftarrow f$(scores of $d$ in each $L_i$)
            $P_i \leftarrow P_i \cup$ {positions seen under sorted or random access} ;
            $bp_i \leftarrow$ best position in $L_i$
        $Y \leftarrow k$ data items with highest score so far ;
        $min\_overall\_score \leftarrow$ smallest overall score of data items in $Y$ ;
        $threshold \leftarrow f$(local scores at position $bp_i$ in each $L_i$) ;
        $j \leftarrow j+1$
end
```

Recall that over this database, TA stops at position 6. ♦

It has been proven that, for any set of sorted lists, BPA stops as early as TA, and its execution cost is never higher than TA [Akbarinia et al., 2007a]. It has also been shown that the execution cost of BPA can be $(m-1)$ times lower than that of TA. Although BPA is quite efficient, it still does redundant work. One of the redundancies with BPA (and also TA) is that it may access some data items several times under sorted access in different lists. For example, a data item that is accessed at a position in a list through sorted access and thus accessed in other lists via random access, may be accessed again in the other lists by sorted access at the next positions. An improved algorithm, BPA2 [Akbarinia et al., 2007a], avoids this and is therefore much more efficient than BPA. It does not transfer the seen positions from list owners to the query originator. Thus, the query originator does not need to maintain the seen positions and their local scores. It also accesses each position in a list at most once. The number of accesses to the lists done by BPA2 can be about $(m-1)$ times lower than that of BPA.

16.3.1.2 Top-k Queries in Unstructured Systems

One possible approach for processing top-k queries in unstructured systems is to route the query to all the peers, retrieve all available answers, score them using the scoring function, and return to the user the k highest scored answers. However, this approach is not efficient in terms of response time and communication cost.

The first efficient solution that has been proposed is that of PlanetP [Cuenca-Acuna et al., 2003], which is an unstructured P2P system. In PlanetP, a content-addressable publish/subscribe service replicates data across P2P communities of up to ten thousand peers. The top-k query processing algorithm works as follows. Given a query Q, the query originator computes a relevance ranking of peers with respect to Q, contacts them one by one in decreasing rank order and asks them to return a set of their top-scored data items together with their scores. To compute the relevance of peers, a global fully replicated index is used that contains term-to-peer mappings. This algorithm has very good performance in moderate-scale systems. However, in a large P2P system, keeping the replicated index up-to-date may hurt scalability.

We describe another solution that was developed within the context of APPA, which is a P2P network-independent data management system [Akbarinia et al., 2006a]. A fully distributed framework to execute top-k queries has been proposed that also addresses the volatility of peers during query execution, and deals with situations where some peers leave the system before finishing query processing. Given a top-k query Q with a specified TTL, the basic algorithm called Fully Decentralized Top-k (FD) proceeds as follows (see Algorithm 16.4).

1. **Query forward.** The query originator forwards Q to the accessible peers whose hop-distance from the query originator is less than TTL.
2. **Local query execution and wait.** Each peer p that receives Q executes it locally: it accesses the local data items that match the query predicate, scores them using a scoring function, selects the k top data items and saves them as well as their scores locally. Then p waits to receive its neighbors' results. However, since some of the neighbors may leave the P2P system and never send a score-list to p, the wait time has a limit that is computed for each peer based on the received TTL, network parameters and peer's local processing parameters.
3. **Merge-and-backward.** In this phase, the top scores are bubbled up to the query originator using a tree-based algorithm as follows. After its wait time has expired, p merges its k local top scores with those received from its neighbors and sends the result to its parent (the peer from which it received Q) in the form of a score-list. In order to minimize network traffic, FD does not bubble up the top data items (which could be large), only their scores and addresses. A score-list is simply a list of k pairs (a,s) where a is the address of the peer owning the data item and s its score.
4. **Data retrieval.** After receiving the score-lists from its neighbors, the query originator forms the final score-list by merging its k local top scores with the

merged score-lists received from its neighbors. Then it directly retrieves the k top data items from the peers that hold them.

Algorithm 16.4: Fully Decentralized Top-k (FD)

```
Input: $Q$: top-k query ;
$f$: scoring function;
$TTL$: time to live;
$w$: wait time
Output: $Y$: list of top-k data items
begin
    At query originator peer
    begin
        send $Q$ to neighbors ;
        $Final\_score\_list \leftarrow$ merge local score lists received from neighbors
        for each peer p in $Final\_score\_list$ do
            $Y \leftarrow$ retrieve top-k data items in $p$
    end
    for each peer that receives $Q$ from a peer $p$ do
        $TTL \leftarrow TTL - 1$ ;
        if $TTL > 0$ then
            send $Q$ to neighbors
        $Local\_score\_list \leftarrow$ extract top-k local scores;
        Wait a time $w$;
        $Local\_score\_list \leftarrow Local\_score\_list \cup$ top-k received scores;
        Send $Local\_score\_list$ to $p$
end
```

The algorithm is completely distributed and does not depend on the existence of certain peers, and this makes it possible to address the volatility of peers during query execution. In particular, the following problems are addressed: peers becoming inaccessible in the merge-and-backward phase; peers that hold top data items becoming inaccessible in the data retrieval phase; late reception of score-lists by a peer after its wait time has expired. The performance evaluation of FD shows that it can achieve major performance gains in terms of communication cost and response time [Akbarinia et al., 2006b].

16.3.1.3 Top-k Queries in DHTs

As we discussed earlier, the main functionality of a DHT is to map a set of keys to the peers of the P2P system and lookup efficiently the peer that is responsible for a given key. This offers efficient and scalable support for exact-match queries.

However, supporting top-k queries on top of DHTs is not easy. A simple solution is to retrieve all tuples of the relations involved in the query, compute the score of each retrieved tuple, and finally return the k tuples whose scores are the highest. However, this solution cannot scale up to a large number of stored tuples. Another solution is to store all tuples of each relation using the same key (e.g., relation's name), so that all tuples are stored at the same peer. Then, top-k query processing can be performed at that central peer using well-known centralized algorithms. However, the peer becomes a bottleneck and a single point of failure.

A solution has been proposed as part of APPA project that is based on TA (see Section 16.3.1.1) and a mechanism that stores the shared data in the DHT in a fully distributed fashion [Akbarinia et al., 2007c]. In APPA, peers can store their tuples in the DHT using two complementary methods: tuple storage and attribute-value storage. With tuple storage, each tuple is stored in the DHT using its identifier (e.g., its primary key) as the storage key. This enables looking up a tuple by its identifier similar to a primary index. Attribute value storage individually stores in the DHT the attributes that may appear in a query's equality predicate or in a query's scoring function. Thus, as in secondary indices, it allows looking up the tuples using their attribute values. Attribute value storage has two important properties: (1) after retrieving an attribute value from the DHT, peers can retrieve easily the corresponding tuple of the attribute value; (2) attribute values that are relatively "close" are stored at the same peer. To provide the first property, the key, which is used for storing the entire tuple, is stored along with the attribute value. The second property is provided using the concept of domain partitioning as follows. Consider an attribute a and let D_a be its domain of values. Assume that there is a total order $<$ on D_a (e.g., D_a is numeric). D_a is partitioned into n non-empty sub-domains $d_1, d_2, \ldots, d_n$ such that their union is equal to D_a, the intersection of any two different sub-domains is empty, and for each $v_1 \in d_i$ and $v_2 \in d_j$, if $i < j$ then we have $v_1 < v_2$. The hash function is applied on the sub-domain of the attribute value. Thus, for the attribute values that fall in the same sub-domain, the storage key is the same and they are stored at the same peer. To avoid attribute storage skew (i.e., skewed distribution of attribute values within sub-domains), domain partitioning is done in such a way that attribute values are uniformly distributed in sub-domains. This technique uses histogram-based information that describes the distribution of values of the attribute.

Using this storage model, the top-k query processing algorithm, called DHTop (see Algorithm 16.5), works as follows. Let Q be a given top-k query, f be its scoring function, and p_0 be the peer at which Q is issued. For simplicity, let us assume that f is a monotonic scoring function. Let scoring attributes be the set of attributes that are passed to the scoring function as arguments. DHTop starts at p_0 and proceeds in two phases: first it prepares ordered lists of candidate sub-domains, and then it continuously retrieves candidate attribute values and their tuples until it finds k top tuples. The details of the two steps are as follows:

1. For each scoring attribute a, p_0 prepares the list of sub-domains and sorts them in descending order of their positive impact on the scoring function. For each list, p_0 removes from the list the sub-domains in which no member can

satisfy Q's conditions. For instance, if there is a condition that enforces the scoring attribute to be equal to a constant, (e.g., $a = 10$), then p_0 removes from the list all the sub-domains except the sub-domain to which the constant value belongs. Let us denote by L_a the list prepared in this phase for a scoring attribute a.

2. For each scoring attribute a, in parallel, p_0 proceeds as follows. It sends Q and a to the peer, say p, that is responsible for storing the values of the first sub-domain of L_a, and requests it to return the values of a at p. The values are returned to p_0 in order of their positive impact on the scoring function. After receiving each attribute value, p_0 retrieves its corresponding tuple, computes its score, and keeps it if the score is one of the k highest scores yet computed. This process continues until k tuples are obtained whose scores are higher than a threshold that is computed based on the attribute values retrieved so far. If the attribute values that p returns to p_0 are not sufficient for determining the k top tuples, p_0 sends Q and a to the site that is responsible for the second sub-domain of L_a and so on until k top tuples are found.

Let $a_1, a_2, \ldots, a_m$ be the scoring attributes and $v_1, v_2, \ldots, v_m$ be the last values retrieved respectively for each of them. The threshold is defined to be $\tau = f(v_1, v_2, \ldots, v_m)$. A main feature of DHTop is that after retrieving each new attribute value, the value of the threshold decreases. Thus, after retrieving a certain number of attribute values and their tuples, the threshold becomes less than k of the retrieved data items and the algorithm stops. It has been analytically proven that DHTop works correctly for monotonic scoring functions and also for a large group of non-monotonic functions.

16.3.1.4 Top-k Queries in Super-peer Systems

A typical algorithm for top-k query processing in super-peer systems is that of Edutella [Balke et al., 2005]. In Edutella, a small percentage of nodes are super-peers and are assumed to be highly available with very good computing capacity. The super-peers are responsible for top-k query processing and the other peers only execute the queries locally and score their resources. The algorithm is quite simple and works as follows. Given a query Q, the query originator sends Q to its super-peer, which then sends it to the other super-peers. The super-peers forward Q to the relevant peers connected to them. Each peer that has some data items relevant to Q scores them and sends its maximum scored data item to its super-peer. Each super-peer chooses the overall maximum scored item from all received data items. For determining the second best item, it only asks one peer, one that has returned the first top item, to return its second top scored item. The super-peer selects the overall second top item from the previously received items and the newly received item. Then, it asks the peer which has returned the second top item and so on until all k top items are retrieved. Finally the super-peers send their top items to the super-peer of the query originator, to extract the overall k top items, and send them to the query originator.

Algorithm 16.5: DHT Top-k (DHTop)

```
Input: $Q$: top-k query;
$f$: scoring function;
$A$: set of $m$ attributes used in $f$
Output: $Y$: list of top-k tuples
begin
    {Phase 1: prepare lists of attributes' subdomains}
    for each scoring attribute $a$ in $A$ do
        $L_a \leftarrow$ all sub-domains of $a$;
        $L_a \leftarrow L_a -$ sub-domains which do not satisfy $Q$'s condition;
        Sort $L_a$ in descending order of its sub-domains
    {Phase 2: continuously retrieve attribute values and their tuples until finding
    k top tuples}
    $Done \leftarrow$ false;
    for each scoring attribute $a$ in $A$ in parallel do
        $i \leftarrow 1$
        while ($i <$ number of sub-domains of $a$) and not $Done$ do
            send $Q$ to peer $p$ that maintains the attribute values of sub-domain $i$
            in $L_a$;
            $Z \leftarrow a$ values (in descending order) from $p$ that satisfy $Q$'s
            condition, along with their corresponding data storage keys ;
            for each received value $v$ do
                get the tuple of $v$;
                $Y \leftarrow k$ tuples with highest score so far;
                $threshold \leftarrow f(v_1, v_2, \ldots, v_m)$ such that $v_i$ is the last value
                received for attribute $a_i$ in $A$;
                $min\_overall\_score \leftarrow$ smallest overall score of tuples in $Y$;
                if $min\_overall\_score \leq threshold$ then
                    $Done \leftarrow$ true
                $i \leftarrow i + 1$
end
```

This algorithm minimizes communication between peers and super-peers since, after having received the maximum scored data items from each peer connected to it, each super-peer asks only one peer for the next top item.

16.3.2 Join Queries

The most efficient join algorithms in distributed and parallel databases are hash-based. Thus, the fact that a DHT relies on hashing to store and locate data can be naturally exploited to support join queries efficiently. A basic solution has been proposed in the context of the PIER P2P system [Huebsch et al., 2003] that provides support for complex queries on top of DHTs. The solution is a variation of the parallel hash join algorithm (PHJ) (see Section 14.3.2) which we call PIERjoin. As in the PHJ algorithm, PIERjoin assumes that the joined relations and the result relations have a home (called *namespace* in PIER), which are the nodes that store horizontal fragments of the relation. Then it makes use of the `put` method for distributing tuples onto a set of peers based on their join attribute so that tuples with the same join attribute values are stored at the same peers. To perform joins locally, PIER implements a version of the symmetric hash join algorithm [Wilschut and Apers, 1991] that provides efficient support for pipelined parallelism. In symmetric hash join, with two joining relations, each node that receives tuples to be joined maintains two hash tables, one per relation. Thus, upon receiving a new tuple from either relation, the node adds the tuple into the corresponding hash table and probes it against the opposite hash table based on the tuples received so far. PIER also relies on the DHT to deal with the dynamic behavior of peers (joining or leaving the network during query execution) and thus does not give guarantees on result completeness.

For a binary join query Q (which may include select predicates), PIERjoin works in three phases (see Algorithm 16.6): multicast, hash and probe/join.

1. **Multicast phase.** The query originator peer multicasts Q to all peers that store tuples of the join relations R and S, i.e., their homes.
2. **Hash phase.** Each peer that receives Q scans its local relation, searching for the tuples that satisfy the select predicate (if any). Then, it sends the selected tuples to the home of the result relation, using `put` operations. The DHT key used in the put operation is calculated using the home of the result relation and the join attribute.
3. **Probe/join phase.** Each peer in the home of the result relation, upon receiving a new tuple, inserts it in the corresponding hash table, probes the opposite hash table to find tuples that match the join predicate (and a select predicate if any) and constructs the result joined tuples. Recall that the "home" of a (horizontally partitioned) relation was defined in Chapter 8 as a set of peers where each peer has a different partition. In this case, the partitioning is by hashing on the join attribute. The home of the result relation is also a partitioned relation (using `put` operations) so it is also at multiple peers.

This basic algorithm can be improved in several ways. For instance, if one of the relations is already hashed on the join attributes, we may use its home as result home, using a variation of the parallel associative join algorithm (PAJ) (see Section 14.3.2), where only one relation needs to be hashed and sent over the DHT.

To avoid multicasting the query to large numbers of peers, another approach is to allocate a limited number of special powerful peers, called *range guards*, for the task of join query processing [Triantafillou and Pitoura, 2003]. The domains of the join attributes are divided, and each partition is dedicated to a range guard. Then, join queries are sent only to range guards, where the query is executed.

Algorithm 16.6: PIERjoin

```
Input: $Q$: join query over relations $R$ and $S$ on attribute $A$;
$h$: hash function;
$H_R, H_S$: homes of $R$ and $S$
Output: $T$: join result relation;
$H_T$: home of $T$
begin
    {Multicast phase}
    At query originator peer send $Q$ to all peers in $H_R$ and $H_S$ ;
    {Hash phase}
    for each peer $p$ in $H_R$ that received $Q$ in parallel do
        for each tuple $r$ in $R_p$ that satisfies the select predicate do
            place $r$ using $h(H_T, A)$
    for each peer $p$ in $H_S$ that received $Q$ in parallel do
        for each tuple $s$ in $S_p$ that satisfies the select predicate do
            place $s$ using $h(H_T, A)$
    {Probe/join phase}
    for each peer $p$ in $H_T$ in parallel do
        if a new tuple $i$ has arrived then
            if $i$ is an $r$ tuple then
                probe $s$ tuples in $S_p$ using $h(A)$
            else
                probe $r$ tuples in $R_p$ using $h(A)$
            $T_p \leftarrow r \bowtie s$
end
```

16.3.3 Range Queries

Recall that range queries have a WHERE clause of the form “attribute A in range $[a,b]$”, with a and b being numerical values. Structured P2P systems, in particular, DHTs are very efficient at supporting exact-match queries (of the form “$A = a$”) but have difficuties with range queries. The main reason is that hashing tends to destroy the ordering of data that is useful in finding ranges quickly.

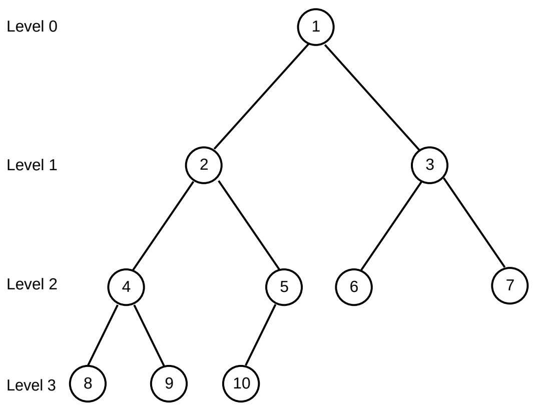
There are two main approaches for supporting range queries in structured P2P systems: extend a DHT with proximity or order-preserving properties, or maintain the key ordering with a tree-based structure. The first approach has been used in several systems. Locality sentitive hashing [Gupta et al., 2003] is an extension to DHTs that hashes similar ranges to the same DHT node with high probability. However, this method can only obtain approximate answers and may cause unbalanced loads in large networks. SkipNet [Harvey et al., 2003] is a lexicographic order-preserving DHT that allows data items with similar values to be placed on contiguous peers. It uses names rather than hashed identifiers to order peers in the overlay network, and each peer is responsible for a range of strings. This facilitates the execution of range queries. However, the number of peers to be visited is linear in the query range.

The Prefix Hash Tree (PHT) [Ramabhadran et al., 2004] is a trie-based distributed data structure that supports range queries over a DHT, by simply using the DHT lookup operation. The data being indexed are binary strings of length D. Each node has either 0 or 2 children, and a key k is stored at a leaf node whose label is a prefix of k. Furthermore, leaf nodes are linked to their neighbors. PHT's lookup operation on key k must return the unique leaf node $leaf(k)$ whose label is a prefix of k. Given a key k of length D, there are $D+1$ distinct prefixes of k. Obtaining $leaf(k)$ can be performed by a linear scan of these potential $D+1$ nodes. However, since a PHT is a binary trie, the linear scan can be improved using a binary search on prefix length. This reduces the number of DHT lookups from $(D+1)$ to $(\log D)$. Given two keys a and b such as $a \leq b$, two algorithms for range queries are supported, using PHT's lookup. The first one is sequential: it searches $leaf(a)$ and then scans sequentially the linked list of leaf nodes until the node $leaf(b)$ is reached. The second algorithm is parallel: it first identifies the node which corresponds to the smallest prefix range that completely covers the range $[a,b]$. To reach this node, a simple DHT lookup is used and the query is forwarded recursively to those children that overlap with the range $[a,b]$.

As in all hashing schemes, the first approach suffers from data skew that can result in peers with unbalanced ranges, which hurts load balancing. To overcome this problem, the second approach exploits tree-based structures to maintain balanced ranges of keys. The first attempt to build a P2P network based on a balanced tree structure is BATON (BAlanced Tree Overlay Network) [Jagadish et al., 2005]. We now present BATON and its support for range queries in more detail.

BATON organizes peers as a balanced binary tree (each node of the tree is maintained by a peer). The position of a node in BATON is determined by a (level,number) tuple, with level starting from 0 at the root, number starting from 1 at the root and sequentially assigned using in-order traversal. Each tree node stores links to its parent, children, adjacent nodes and selected neighbor nodes that are nodes at the same level. Two routing tables: a *left routing table* and a *right routing table* store links to the selected neighbor nodes. For a node numbered i, these routing tables contain links to nodes located at the same level with numbers that are less (left routing table) and greater (right routing table) than i by a power of 2. The j^{th} element in the left (right) routing table at node i contains a link to the node numbered $i-2^{j-1}$ (respectively $i+2^{j-1}$) at the same level in the tree. Figure 16.11 shows the routing table of node 6.

In BATON, each leaf and internal node (or peer) is assigned a range of values. For each link this range is stored at the routing table and when its range changes, the link is modified to record the change. The range of values managed by a peer is required to be to the right of the range managed by its left subtree and less than the range managed by its right subtree (see Figure 16.12). Thus, BATON builds an effective distributed index structure. The joining and departure of peers are processed such that the tree remains balanced by forwarding the request upward in the tree for joins



Node 6: level 2, number=3
parent=3, leftchild=null, rightchild=null
leftadjacent=1, rightadjacent=3

Left routing table

	Node	Left Child	Right Child	Lower Bound	Upper Bound
0	5	10	null	LB5	UB5
1	4	8	9	LB4	UB4

Right routing table

	Node	Left Child	Right Child	Lower Bound	Upper Bound
0	7	null	null	LB7	UB7

Fig. 16.11 BATON structure-tree index and routing table of node 6

and downward in the tree for leaves, thus with no more than $O(\log\ n)$ steps for a tree of n nodes.

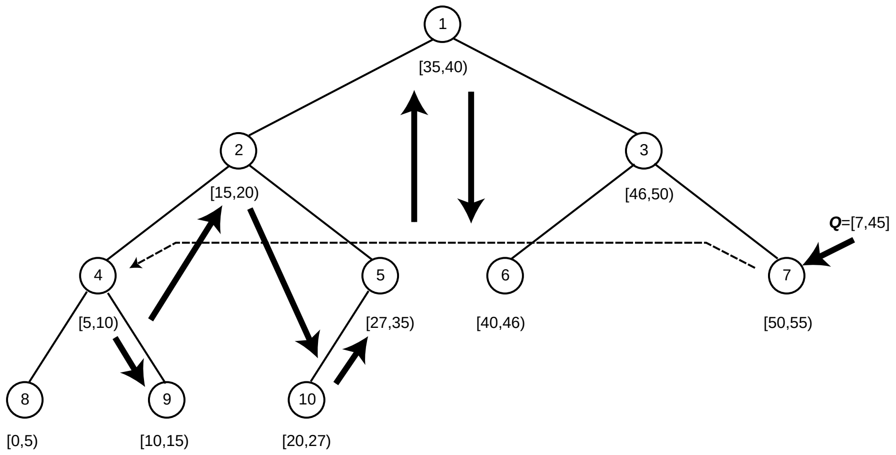


Fig. 16.12 Range query processing in BATON

A range query is processed as follows (Algorithm 16.7). For a range query Q with range $[a,b]$ submitted by node i, it looks for a node that intersects with the lower bound of the searched range. The peer that stores the lower bound of the range checks locally for tuples belonging to the range and forwards the query to its right adjacent node. In general, each node receiving the query checks for local tuples and contacts its right adjacent node until the node containing the upper bound of the range is reached. Partial answers obtained when an intersection is found are sent to the node that submits the query. The first intersection is found in $O(\log\ n)$ steps using an algorithm for exact match queries. Therefore, a range query with X nodes covering the range is answered in $O(\log n + X)$ steps.

Algorithm 16.7: BatonRange

```
Input: $Q$: a range query in the form $[a,b]$
Output: $T$: result relation
begin
    {Search for the peer storing the lower bound of the range}
    At query originator peer
    begin
        find peer $p$ that holds value $a$ ;
        send $Q$ to $p$;
    end
    for each peer $p$ that receives $Q$ do
        $T_p \leftarrow Range(p) \cap [a,b]$;
        send $T_p$ to query originator ;
        if $Range(RightAdjacent(p)) \cap [a,b] \neq \emptyset$ then
            let $p$ be right adjacent peer of $p$ ;
            send $Q$ to $p$
end
```

Example 16.6. Consider the query Q with range $[7,45]$ issued at node 7 in Figure 16.12. First, BATON executes an exact match query looking for a node containing the lower bound of the range (see dashed line in the figure). Since the lower bound is in the range assigned to node 4, it checks locally for tuples belonging to the range and forwards the query to its adjacent right node (node 9). Node 9 checks for local tuples belonging to the range and forwards the query to node 2. Nodes 10, 5, 1 and 6 receive the query, they check for local tuples and contact their respective right adjacent node until the node containing the upper bound of the range is reached. ♦

16.4 Replica Consistency

To increase data availability and access performance, P2P systems replicate data. However, different P2P systems provide very different levels of replica consistency. The earlier, simple P2P systems such as Gnutella and Kazaa deal only with static data (e.g., music files) and replication is "passive" as it occurs naturally as peers request and copy files from one another (basically, caching data). In more advanced P2P systems where replicas can be updated, there is a need for proper replica management techniques. Unfortunately, most of the work on replica consistency has been done only in the context of DHTs. We can distinguish three approaches to deal with replica consistency: basic support in DHTs, data currency in DHTs, and replica reconciliation. In this section, we introduce the main techniques used in these approaches.

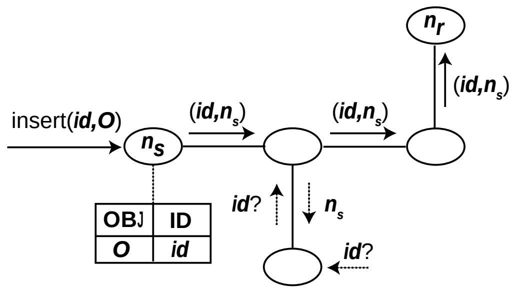
16.4.1 Basic Support in DHTs

To improve data availability, most DHTs rely on data replication by storing $(key, data)$ pairs at several peers by, for example, using several hash functions. If one peer is unavailable, its data can still be retrieved from the other peers that hold a replica. Some DHTs provide basic support for the application to deal with replica consistency. In this section, we describe the techniques used in two popular DHTs: CAN and Tapestry.

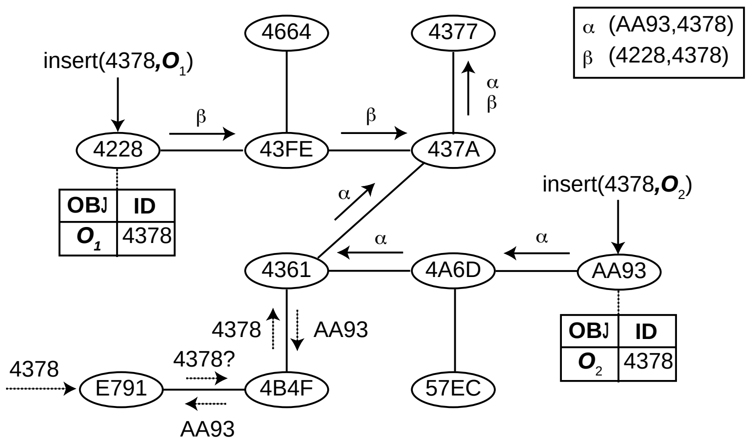
CAN provides two approaches for supporting replication [Ratnasamy et al., 2001a]. The first one is to use m hash functions to map a single key onto m points in the coordinate space, and, accordingly, replicate a single $(key, data)$ pair at m distinct nodes in the network. The second approach is an optimization over the basic design of CAN that consists of a node proactively pushing out popular keys towards its neighbors when it finds it is being overloaded by requests for these keys. In this approach, replicated keys should have an associated TTL field to automatically undo the effect of replication at the end of the overloaded period. In addition, the technique assumes immutable (read-only) data.

Tapestry [Zhao et al., 2004] is an extensible P2P system that provides decentralized object location and routing on top of a structured overlay network. It routes messages to logical end-points (i.e., endpoints whose identifiers are not associated with physical location), such as nodes or object replicas. This enables message delivery to mobile or replicated endpoints in the presence of instability of the underlying infrastructure. In addition, Tapestry takes latency into account to establish each node's neighborhood. The location and routing mechanisms of Tapestry work as follows. Let o be an object identified by $id(o)$; the insertion of o in the P2P network involves two nodes: the server node (noted n_s) that holds o and the root node (noted n_r) that holds a mapping in the form $(id(o), n_s)$ indicating that the object identified by $id(o)$ is stored at node n_s. The root node is dynamically determined by a globally consistent deterministic algorithm. Figure 16.13a shows that when o is inserted into n_s, n_s publishes $id(o)$ at its root node by routing a message from n_s to n_r containing the mapping $(id(o), n_s)$. This mapping is stored at all nodes along the message path. During a location query (e.g., "$id(o)$?" in Figure 16.13a, the message that looks for $id(o)$ is initially routed towards n_r, but it may be stopped before reaching it once a node containing the mapping $(id(o), n_s)$ is found. For routing a message to $id(o)$'s root, each node forwards this message to its neighbor whose logical identifier is the most similar to $id(o)$ [Plaxton et al., 1997].

Tapestry offers the entire infrastructure needed to take advantage of replicas, as shown in Figure 16.13b. Each node in the graph represents a peer in the P2P network and contains the peer's logical identifier in hexadecimal format. In this example, two replicas O_1 and O_2 of object O (e.g., a book file) are inserted into distinct peers



(a) Object publishing



(b) Replica management

Fig. 16.13 Tapestry (a) Object publishing (b) Replica management.

($O_1 \rightarrow$ peer 4228 and $O_2 \rightarrow$ peer $AA93$). The identifier of O_1 is equal to that of O_2 (i.e., 4378 in hexadecimal) as O_1 and O_2 are replicas of the same object O. When O_1 is inserted into its server node (peer 4228), the mapping $(4378, 4228)$ is routed from peer 4228 to peer 4377 (the root node for O_1's identifier). As the message approaches the root node, the object and the node identifiers become increasingly similar. In addition, the mapping $(4378, 4228)$ is stored at all peers along the message path. The insertion of O_2 follows the same procedure. In Figure 16.13b, if peer E791 looks for a replica of O, the associated message routing stops at peer 4361. Therefore, applications can replicate data across multiple server nodes and rely on Tapestry to direct requests to nearby replicas.

16.4.2 Data Currency in DHTs

Although DHTs provide basic support for replication, the mutual consistency of the replicas after updates can be compromised as a result of peers leaving the network or concurrent updates. Let us illustrate the problem with a simple update scenario in a typical DHT.

Example 16.7. Let us assume that the operation `put` (k, d_0) (issued by some peer) maps onto peers p_1 and p_2 both of which get to store data d_0. Now consider an update (from the same or another peer) with the operation `put` (k, d_1) that also maps onto peers p_1 and p_2. Assuming that p_2 cannot be reached (e.g., because it has left the network), only p_1 gets updated to store d_1. When p_2 rejoins the network later on, the replicas are not consistent: p_1 holds the current state of the data associated with k while p_2 holds a stale state.

Concurrent updates also cause problems. Consider now two updates `put` (k, d_2) and `put` (k, d_3) (issued by two different peers) that are sent to p_1 and p_2 in reverse order, so that p_1's last state is d_2 while p_2's last state is d_3. Thus, a subsequent `get` (k) operation will return either stale or current data depending on which peer is looked up, and there is no way to tell whether it is current or not. ♦

For some applications (e.g., agenda management, bulletin boards, cooperative auction management, reservation management, etc.) that could take advantage of a DHT, the ability to get the current data are very important. Supporting data currency in replicated DHTs requires the ability to return a current replica despite peers leaving the network or concurrent updates. Of course, replica consistency is a more general problem, as discussed in Chapter 13, but the issue is particularly difficult and important in P2P systems, since there is considerable dynamism in the peers joining and leaving the system. The problem can be partially addressed by using data versioning [Knezevic et al., 2005]. Each replica has a version number that is increased after each update. To return a current replica, all replicas need to be retrieved in order to select the latest version. However, because of concurrent updates, it may happen that two different replicas have the same version number, thus making it impossible to decide which one is the current replica.

A more complete solution has been proposed that considers both data availability and data currency [Akbarinia et al., 2007b]. To provide high data availability, data are replicated in the DHT using a set of independent hash functions H_r, called *replication hash functions*. The peer that is responsible for key k with respect to hash function h at the current time is denoted by $rsp(k, h)$. To be able to retrieve a current replica, each pair $(k, data)$ is stamped with a logical timestamp, and for each $h \in H_r$, the pair $(k, newData)$ is replicated at $rsp(k, h)$ where $newData = \{data, timestamp\}$, i.e., newdata is composed of the initial data and the timestamp. Upon a request for the data associated with a key, we can return one of the replicas that are stamped with the latest timestamp. The number of replication hash functions, i.e., H_r, can be different for different DHTs. For instance, if in a DHT the availability of peers is low, a high value of H_r (e.g., 30) can be used to increase data availability.

This solution is the basis for a service called *Update Management Service* (UMS) that deals with efficient insertion and retrieval of current replicas based on timestamping. Experimental validation has shown that UMS incurs very little overhead in terms of communication cost. After retrieving a replica, UMS detects whether it is current or not, i.e., without having to compare with the other replicas, and returns it as output. Thus, UMS does not need to retrieve all replicas to find a current one; it only requires the DHT's lookup service with `put` and `get` operations.

To generate timestamps, UMS uses a distributed service called *Key-based Timestamping Service* (KTS). The main operation of KTS is `gen_ts(`k`)`, which, given a key k, generates a real number as a timestamp for k. The timestamps generated by KTS are *monotonic* such that if ts_i and ts_j are two timestamps generated for the same key at times t_i and t_j, respectively, $ts_j > ts_i$ if t_j is later than t_i. This property allows ordering the timestamps generated for the same key according to the time at which they have been generated. KTS has another operation denoted by *last_ts(k)*, which, given a key k, returns the last timestamp generated for k by KTS. At anytime, `gen_ts(`k`)` generates at most one timestamp for k, and different timestamps for k are monotonic. Thus, in the case of concurrent calls to insert a pair $(k, data)$, i.e., from different peers, only the one that obtains the latest timestamp will succeed to store its data in the DHT.

16.4.3 Replica Reconciliation

Replica reconciliation goes one step further than data currency by enforcing mutual consistency of replicas. Since a P2P network is typically very dynamic, with peers joining or leaving the network at will, eager replication solutions (see Chapter 13) are not appropriate; lazy replication is preferred. In this section, we describe the reconciliation techniques used in OceanStore, P-Grid and APPA to provide a spectrum of proposed solutions.

16.4.3.1 OceanStore

OceanStore [Kubiatowicz et al., 2000] is a data management system designed to provide continuous access to persistent information. It relies on Tapestry and assumes an infrastructure composed of untrusted powerful servers that are connected by high-speed links. For security reasons, data are protected through redundancy and cryptographic techniques. To improve performance, data are allowed to be cached anywhere, anytime.

OceanStore allows concurrent updates on replicated objects; it relies on reconciliation to assure data consistency. Figure 16.14 illustrates update management in OceanStore. In this example, R is a replicated object whereas R_i and r_i denote, respectively, a primary and a secondary copy of R. Nodes n_1 and n_2 are concurrently updating R. Such updates are managed as follows. Nodes that hold primary copies of

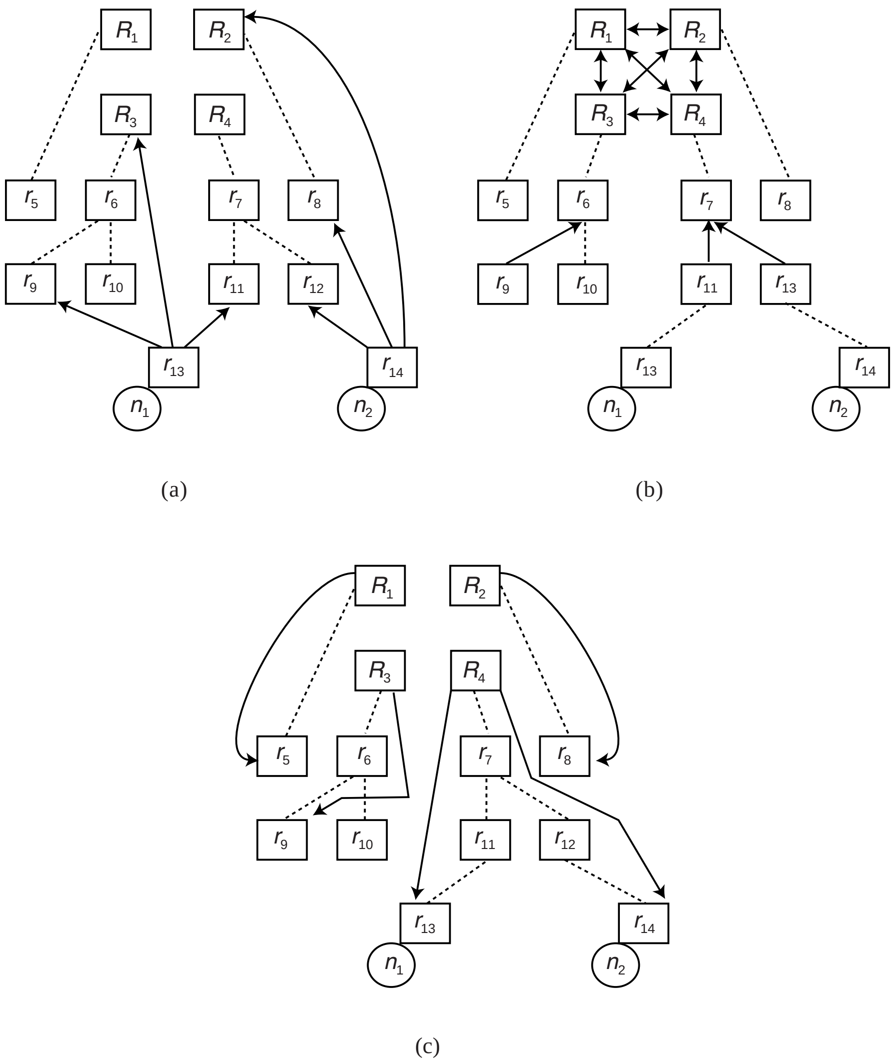


Fig. 16.14 OceanStore reconciliation. (a) Nodes n_1 and n_2 send updates to the master group of R and to several random secondary replicas. (b) The master group of R orders updates while secondary replicas propagate them epidemically. (c) After the master group agreement, the result of updates is multicast to secondary replicas.

R, called the *master group of* R, are responsible for ordering updates. So, n_1 and n_2 perform tentative updates on their local secondary replicas and send these updates to the master group of R as well as to other random secondary replicas (see Figure 16.14a). The tentative updates are ordered by the master group based on timestamps assigned by n_1 and n_2; at the same time, these updates are epidemically propagated among secondary replicas (Figure 16.14b). Once the master group obtains an agreement, the result of updates is multicast to secondary replicas (Figure 16.14c), which contain both tentative[2] and committed data.

Replica management adjusts the number and location of replicas in order to service requests more efficiently. By monitoring the system load, OceanStore detects when a replica is overwhelmed and creates additional replicas on nearby nodes to alleviate load. Conversely, these additional replicas are eliminated when they are no longer needed.

16.4.3.2 P-Grid

P-Grid [Aberer et al., 2003a] is a structured P2P network based on a binary trie structure. A decentralized and self-organizing process builds P-Grid's routing infrastructure which is adapted to a given distribution of data keys stored by peers. This process addresses uniform load distribution of data storage and uniform replication of data to support availability.

To address updates of replicated objects, P-Grid employs gossiping, without strong consistency guarantees. P-Grid assumes that quasi-consistency of replicas (instead of full consistency which is too hard to provide in a dynamic environment) is enough.

The update propagation scheme has a push phase and a pull phase. When a peer p receives a new update to a replicated object R, it pushes the update to a subset of peers that hold replicas of R, which, in turn, propagate it to other peers holding replicas of R, and so on. Peers that have been disconnected and get connected again, peers that do not receive updates for a long time, or peers that receive a pull request but are not sure whether they have the latest update, enter the pull phase to reconcile. In this phase, multiple peers are contacted and the most up-to-date among them is chosen to provide the object content.

16.4.3.3 APPA

APPA provides a general lazy distributed replication solution that assures eventual consistency of replicas [Martins et al., 2006a; Martins and Pacitti, 2006; Martins et al., 2008]. It uses the action-constraint framework [Kermarrec et al., 2001] to capture the application semantics and resolve update conflicts.

The application semantics is described by means of constraints between update actions. An *action* is defined by the application programmer and represents an application-specific operation (e.g., a write operation on a file or document, or a database transaction). A *constraint* is the formal representation of an application invariant. For instance, the *predSucc*(a_1, a_2) constraint establishes causal ordering between actions (i.e., action a_2 executes only after a_1 has succeeded); the *mutuallyExclusive*(a_1, a_2) constraint states that either a_1 or a_2 can be executed. The aim of reconciliation is to take a set of actions with the associated constraints and produce

[2] Tentative data are data that the primary replicas have not yet committed.

a *schedule*, i.e., a list of ordered actions that do not violate constraints. In order to reduce the schedule production complexity, the set of actions to be ordered is divided into subsets called *clusters*. A cluster is a subset of actions related by constraints that can be ordered independently of other clusters. Therefore, the *global schedule* is composed by the concatenation of clusters' ordered actions.

Data managed by the APPA reconciliation algorithm are stored in data structures called *reconciliation objects*. Each reconciliation object has a unique identifier in order to enable its storage and retrieval in the DHT. Data replication proceeds as follows. First, nodes execute local actions to update a replica of an object while respecting user-defined constraints. Then, these actions (with the associated constraints) are stored in the DHT based on the object's identifier. Finally, reconciler nodes retrieve actions and constraints from the DHT and produce the global schedule, by reconciling conflicting actions based on the application semantics. This schedule is locally executed at every node, thereby assuring eventual consistency.

Any connected node can try to start reconciliation by inviting other available nodes to engage with it. Only one reconciliation can run at-a-time. The reconciliation of update actions is performed in 6 distributed steps as follows. Nodes at step 2 start reconciliation. The outputs produced at each step become the input to the next one.

- **Step 1 - node allocation:** a subset of connected replica nodes is selected to proceed as reconcilers based on communication costs.
- **Step 2 - action grouping:** reconcilers take actions from the action logs and put actions that try to update common objects into the same group since these actions are potentially in conflict. Groups of actions that try to update object R are stored in the *action log R* reconciliation object (L_R).
- **Step 3 - cluster creation:** reconcilers take action groups from the action logs and split them into clusters of semantically dependent conflicting actions (two actions a_1 and a_2 are semantically independent if the application judges it safe to execute them together, in any order, even if they update a common object; otherwise, a_1 and a_2 are semantically dependent. Clusters produced in this step are stored in the cluster set reconciliation object.
- **Step 4 - clusters extension:** user-defined constraints are not taken into account in cluster creation. Thus, in this step, reconcilers extend clusters by adding to them new conflicting actions, according to user-defined constraints.
- **Step 5 - cluster integration:** cluster extensions lead to cluster overlapping (an overlap occurs when the intersection of two clusters results in a non-null set of actions). In this step, reconcilers bring together overlapping clusters. At this point, clusters become mutually-independent, i.e., there are no constraints involving actions of distinct clusters.
- **Step 6 - cluster ordering:** in this step, reconcilers take each cluster from the cluster set and order the cluster's actions. The ordered actions associated with each cluster are stored in the *schedule* reconciliation object. The concatenation of all clusters' ordered actions makes up the global schedule that is executed by all replica nodes.

At every step, the reconciliation algorithm takes advantage of data parallelism, i.e., several nodes per-form simultaneously independent activities on a distinct subset of actions (e.g., ordering of different clusters).

16.5 Conclusion

By distributing data storage and processing across autonomous peers in the network, “modern” P2P systems can scale without the need for powerful servers. Advanced P2P applications such as scientific cooperation must deal with semantically rich data (e.g., XML documents, relational tables, etc.). Supporting such applications requires significant revisiting of distributed database techniques (schema management, access control, query processing, transaction management, consistency management, reliability and replication). When considering data management, the main requirements of a P2P system are autonomy, query expressiveness, efficiency, quality of service, and fault-tolerance. Depending on the P2P network architecture (unstructured, structured DHT, or hybrid super-peer), these requirements can be achieved to varying degrees. Unstructured networks have better fault-tolerance but can be quite inefficient because they rely on flooding for query routing. Hybrid systems have better potential to satisfy high-level data management requirements. However, DHT systems are best for key-based search and could be combined with super-peer networks for more complex searching.

Most of the work on sharing semantically rich data in P2P systems has focused on schema management and query processing. However, there has been very little work on update management, replication, transactions and access control. Much more work is needed to revisit distributed database techniques for large-scale P2P systems. The main issues that have to be dealt with include schema management, complex query processing, transaction support and replication, and privacy. Furthermore, it is unlikely that all kinds of data management applications are suited for P2P systems. Typical applications that can take advantage of P2P systems are probably light-weight and involve some sort of cooperation. Characterizing carefully these applications is important and will be useful to produce performance benchmarks.

16.6 Bibliographic Notes

Data management in “modern” P2P systems, those characterized by massive distribution, inherent heterogeneity, and high volatility, has become an important research topic. The topic is fully covered in a recent book [Vu et al., 2009]. A shorter survey can be found in [Ulusoy, 2007]. Discussions on the requirements, architectures, and issues faced by P2P data management systems are provided in [Bernstein et al., 2002; Daswani et al., 2003; Valduriez and Pacitti, 2004]. A number of P2P data management systems are presented in [Aberer, 2003].

An extensive survey of query processing in P2P systems is provided in [Akbarinia et al., 2007d] and has been the basis for writing Sections 16.2 and 16.3. A good discussion of the issues of schema mapping in P2P systems can be found in [Tatarinov et al., 2003]. An important kind of query in P2P systems is top-k queries. A survey of top-k query processing techniques in relational database systems is provided in [Ilyas et al., 2008]. An efficient algorithm for top-k query processing is the Threshold Algorithm (TA) which was proposed independently by several researchers [Nepal and Ramakrishna, 1999; Güntzer et al., 2000; Fagin et al., 2003]. TA has been the basis for several algorithms in P2P systems, in particular in DHTs [Akbarinia et al., 2007c]. A more efficient algorithm than TA is the Best Position Algorithm [Akbarinia et al., 2007a]. A survey of ranking algorithms in databases (not necessarily in P2P systems) is given in [Ilyas et al., 2008].

The survey of replication in P2P systems by Martins et al. [2006b] has been the basis for Section 16.4. A complete solution to data currency in replicated DHTs, i.e., providing the ability to find the most current replica, is given in [Akbarinia et al., 2007b]. Reconciliation of replicated data are addressed in OceanStore [Kubiatowicz et al., 2000], P-Grid [Aberer et al., 2003a] and APPA [Martins et al., 2006a; Martins and Pacitti, 2006].

P2P techniques have recently received attention to help scaling up data management in the context of Grid Computing. This triggered open problems and new issues which are discussed in [Pacitti et al., 2007a].

Exercises

Problem 16.1. What is the fundamental difference between P2P and client-server architectures? Is a P2P system with a centralized index equivalent to a client-server system? List the main advantages and drawbacks of P2P file sharing systems from different points of view:

- end-users;
- file owners;
- network administrators.

Problem 16.2 (**). A P2P overlay network is built as a layer on top of a physical network, typically the Internet. Thus, they have different topologies and two nodes that are neighbors in the P2P network may be far apart in the physical network. What are the advantages and drawbacks of this layering? What is the impact of this layering on the design of the three main types of P2P networks (unstructured, structured and superpeer)?

Problem 16.3 (*). Consider the unstructured P2P network in Figure 16.4 and the bottom-left peer that sends a request for resource. Illustrate and discuss the two following search strategies in terms of result completeness:

- flooding with TTL=3;
- gossiping with each peer has a partial view of at most 3 neighbours.

Problem 16.4 (*). Consider Figure 16.7, focusing on structured networks. Refine the comparison using the scale 1-5 (instead of low - moderate - high) by considering the three main types of DHTs: tree, hypercube and ring.

Problem 16.5 (**). The objective is to design a P2P social network application, on top of a DHT. The application should provide basic functions of social networks: register a new user with her profile; invite or retrieve friends; create lists of friends; post a message to friends; read friends' messages; post a comment on a message. Assume a generic DHT with put and get operations, where each user is a peer in the DHT.

Problem 16.6 (**). Propose a P2P architecture of the social network application, with the (key, data) pairs for the different entities which need be distributed. Describe how the following operations: create or remove a user; create or remove a friendship; read messages from a list of friends. Discuss the advantages and drawbacks of the design.

Problem 16.7 (**). Same question, but with the additional requirement that private data (e.g., user profile) must be stored at the user peer.

Problem 16.8. Discuss the commonalities and differences of schema mapping in multidatabase systems and P2P systems. In particular, compare the local-as-view approach presented in Chapter 4 with the pairwise schema mapping approach in Section 16.2.1.

Problem 16.9 (*). The FD algorithm for top-k query processing in unstructured P2P networks (see Algorithm 16.4) relies on flooding. Propose a variation of FD where, instead of flooding, random walk or gossiping is used. What are the advantages and drawbacks?

Problem 16.10 (*). Apply the TPUT algorithm (Algorithm 16.2) to the three lists of the database in Figure 16.10 witk k=3. For each step of the algorithm, show the intermediate results.

Problem 16.11 (*). Same question applied to Algorithm DHTop (see Algorithm 16.5.

Problem 16.12 (*). Algorithm 16.6 assumes that the input relations to be joined are placed arbitrarily in the DHT. Assuming that one of the relations is already hashed on the join attributes, propose an improvement of Algorithm 16.6.

Problem 16.13 (*). To improve data availability in DHTs, a common solution is to replicate $(k, data)$ pairs at several peers using several hash functions. This produces the problem illustrated in Example 16.7. An alternative solution is to use a non-replicated DHT (with a single hash function) and have the nodes replicating (k, data) pairs at some of their neighbors. What is the effect on the scenario in Example 16.7? What are the advantages and drawbacks of this approach, in terms of availability and load balancing?

Chapter 17
Web Data Management

The World Wide Web ("WWW" or "web" for short) has become a major repository of data and documents. Although measurements differ and change, the web has grown at a phenomenal rate. According to two studies in 1998, there were 200 million [Bharat and Broder, 1998] to upwards of 320 million [Lawrence and Giles, 1998] static web pages. A 1999 study reported the size of the web as 800 million pages [Lawrence and Giles, 1999]. By 2005, the number of pages were reported to be 11.5 billion [Gulli and Signorini, 2005]. Today it is estimated that the web contains over 25 billion pages[1] and growing. These are numbers for the "static" web pages, i.e., those whose content do not change unless the page owners make explicit changes. The size of the web is much larger when "dynamic" web pages (i.e., pages whose content changes based on the context of user requests) are considered. A 2005 study reported the size to be over 53 billion pages [Hirate et al., 2006]. Additionally, it was estimated that, as of 2001, over 500 billion documents existed in the *deep web* (which we define below) [Bergman, 2001]. Besides its size, the web is very dynamic and changes rapidly. Thus, for all practical purposes, the web represents a very large, dynamic and distributed data store and there are the obvious distributed data management issues in accessing web data.

The web, in its present form, can be viewed as two distinct yet related components. The first of these components is what is known as the *publicly indexable web* (PIW) [Lawrence and Giles, 1998]. This is composed of all static (and cross-linked) web pages that exist on web servers. The other component, which is known as the *hidden web* [Florescu et al., 1998] (or the *deep web* [Raghavan and Garcia-Molina, 2001]), is composed of a huge number of databases that encapsulate the data, hiding it from the outside world. The data in the hidden web are usually retrieved by means of search interfaces where the user enters a query that is passed to the database server, and the results are returned to the user as a dynamically generated web page.

The difference between the two is basically in the way they are handled for searching and/or querying. Searching the PIW depends mainly on crawling its pages using the link structure between them, indexing the crawled pages, and then

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

searching the indexed data (as we discuss at length in Section 17.2). It is not possible to apply this approach to the hidden web directly since it is not possible to crawl and index those data (the techniques for searching the hidden web are discussed in Section 17.3.4).

Research on web data management has followed different threads. Most of the earlier work focused on keyword search and search engines. The subsequent work in the database community focused on declarative querying of web data. There is an emerging trend that combines search/browse mode of access with declarative querying, but this work has not yet reached its full potential. Along another front, XML has emerged as an important data format for representing data on the web. Thus, XML data management, and more recently *distributed* XML data management, have been topics of interest. The result of these different threads of development is that there is little in the way of a unifying architecture or framework for discussing web data management, and the different lines of research have to be considered somewhat separately. Furthermore, the full coverage of all the web-related topics requires far deeper and far more extensive treatment than is possible within a chapter. Therefore, we focus on issues that are directly related to data management.

We start by discussing how web data can be modelled as a graph. Both the structure of this graph and its management are important. This is discussed in Section 17.1. Web search is discussed in Section 17.2 and web querying is covered in Section 17.3. These are fundamental topics in web data management. We then discuss distributed XML data management (Section 17.4). Although the web pages were originally encoded using HTML, the use of XML and the prevalence of XML-encoded data are increasing, particularly in the data repositories available on the web. Therefore, the distributed management of XML data is increasingly important.

17.1 Web Graph Management

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

(a) It is quite volatile. We already discussed the speed with which the graph is growing. In addition, a significant proportion of the web pages experience frequent updates.

(b) It is sparse. A graph is considered sparse if its average degree is less than the number of vertices. This means that the each node of the graph has a limited number of neighbors, even if the nodes are in general connected. The sparseness of the web graph implies an interesting graph structure that we discuss shortly.

(c) It is "self-organizing." The web contains a number of communities, each of which consist of a set of pages that focus on a particular topic. These communities get organized on their own without any "centralized control," and give rise to the particular subgraphs in the web graph.

(d) It is a "small-world network." This property is related to sparseness – each node in the graph may not have many neighbors (i.e., its degree may be small), but many nodes are connected through intermediaries. Small-world networks were first identified in social sciences where it was noted that many people who are strangers to each other are connected by intermediaries. This holds true in web graphs as well in terms of the connectedness of the graph.

(e) It is a power law network. The in- and out-degree distributions of the web graph follow power law distributions. This means that the probability that a node has in- (out-) degree i is proportional to $1/i^{\alpha}$ for some $\alpha > 1$.The value of α is about 2.1 for in-degree and about 7.2 for out-degree [Broder et al., 2000].

This brings us to a discussion of the structure of the web graph, which has a "bowtie" shape (Figure 17.1) [Broder et al., 2000]. It has a strongly connected component (the knot in the middle) in which there is a path between each pair of pages. The strongly connected component (SCC) accounts for about 28% of the web pages. A further 21% of the pages constitute the "IN" component from which there are paths to pages in SCC, but to which no paths exist from pages in SCC. Symmetrically, "OUT" component has pages to which paths exists from pages in SCC but not vice versa, and these also constitute 21% of the pages. What is referred to as "tendrils" consist of pages that cannot be reached from SCC and from which SCC pages cannot be reached either. These constitute about 22% of the web pages. These are pages that have not yet been "discovered" and have not yet been connected to the better known parts of the web. Finally, there are disconnected components that have no links to/from anything except their own small communities. This makes up about 8% of the web. This structure is interesting in that it determines the results that one gets from web searches and from querying the web. Furthermore, this graph structure is different than many other graphs that are normally studied, requiring special algorithms and techniques for its management.

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

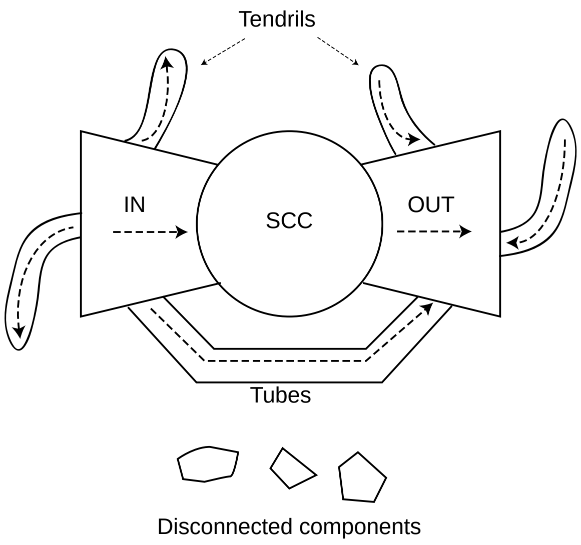


Fig. 17.1 The Structure of the web as a Bowtie (Based on [Kumar et al., 2000].)

17.1.1 Compressing Web Graphs

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

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

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

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

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

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

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

17.1.2 Storing Web Graphs as S-Nodes

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

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

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

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

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

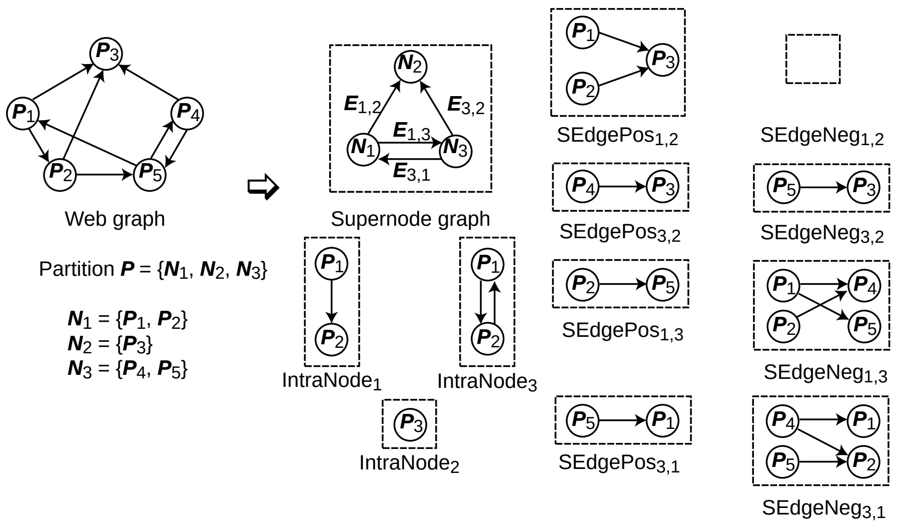


Fig. 17.2 Partitioning the web graph (Based on [Raghavan and Garcia-Molina, 2003].)

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

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

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

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

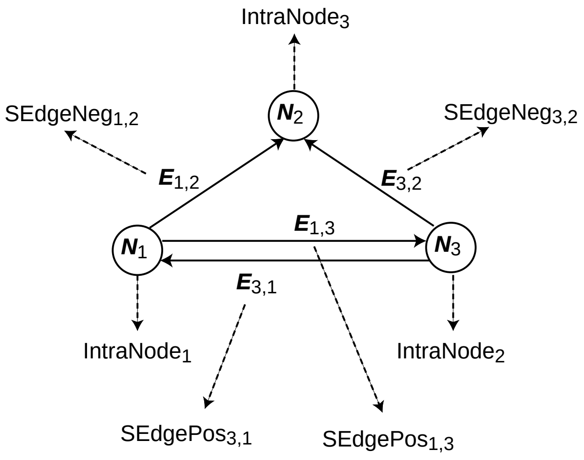


Fig. 17.3 S-node representation (Based on [Raghavan and Garcia-Molina, 2003].)

17.2 Web Search

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

The abstract architecture of a generic search engine is shown in Figure17.4 [Arasu et al., 2001]. We discuss the components of this architecture in some detail.

In every search engine the *crawler* plays one of the most crucial roles. A crawler is a program used by a search engine to scan the web on its behalf and collect data about web pages. A crawler is given a starting set of pages – more accurately, it is given a set of Uniform Resource Locators (URLs) that identify these pages. The crawler retrieves and parses the page corresponding to that URL, extracts any URLs in it, and adds these URLs to a queue. In the next cycle, the crawler extracts a URL from the queue (based on some order) and retrieves the corresponding page. This process is repeated until the crawler stops. A control module is responsible for deciding which URLs should be visited next. The retrieved pages are stored in a page repository. Section 17.2.1 examines crawling operations in more detail.

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

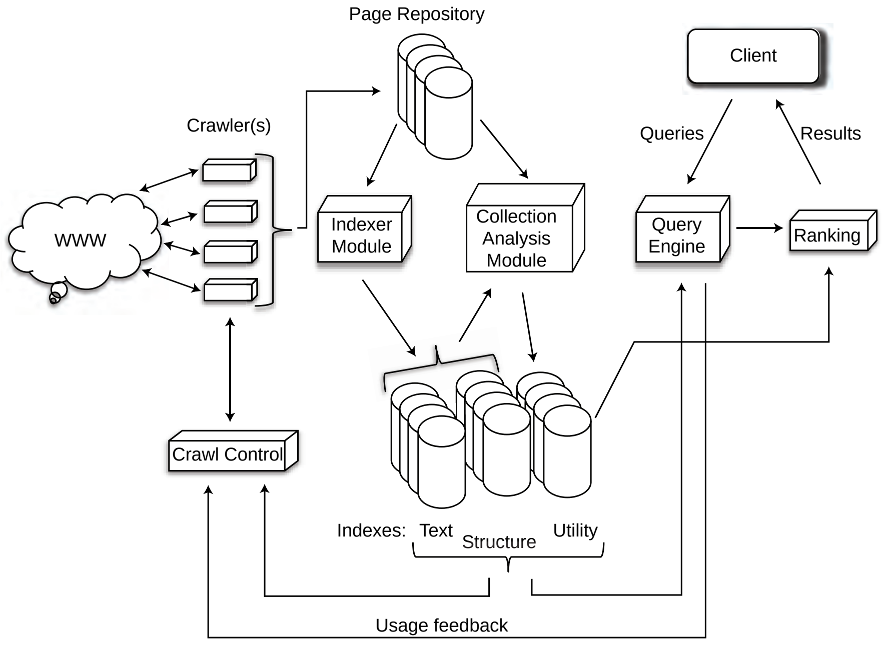


Fig. 17.4 Search Engine Architecture (Based on [**?**]

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

The *ranking module* is responsible for sorting the large number of results so that those that are considered to be most relevant to the user's search are presented first. The problem of ranking has drawn increased interest in order to go beyond traditional information retrieval (IR) techniques to address the special characteristics of the web — web queries are usually small and they are executed over a vast amount of data. Section 17.2.3 introduces algorithms for ranking and describes approaches that exploit the link structure of the web to obtain improved ranking results.

17.2.1 Web Crawling

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

There are a number of issues that need to be addressed in designing a crawler [Cho et al., 1998]. Since the primary goal is to access more important pages before others, there needs to be some way of determining the importance of a page. This can be done by means of a measure that reflects the importance of a given page. These measures can be static, such that the importance of a page is determined independent of retrieval queries that will run against it, or dynamic in that they take the queries into consideration. Examples of static measures are those that determine the importance of a page P with respect to the number of pages that point to P (referred to as *backlink*), or those that additionally take into account the importance of the backlink pages as is done in the popular PageRank metric [Page et al., 1998] that is used by Google and others. A possible dynamic measure may be one that calculates the importance of a page P with respect its textual similarity to the query that is being evaluated using some of the well-known information retrieval similarity measures.

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

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

This formula calculates the rank of a page based on the backlinks, but normalizes the contribution of each backlinking page P_j using the number of links that P_j has to other pages. The idea here is that it is more important to be pointed at by pages conservatively link to other pages than by those who link to others indiscriminately.

A second issue is how the crawler chooses the next page to visit once it has crawled a particular page. As noted earlier, the crawler maintains a queue in which it stores the URLs for the pages that it discovers as it analyzes each page. Thus, the issue is one of ordering the URLs in this queue. A number of strategies are possible. One possibility is to visit the URLs in the order in which they were discovered; this is referred to as the *breadth-first approach* [Cho et al., 1998; Najork and Wiener, 2001]. Another alternative is to use random ordering whereby the crawler chooses a URL randomly from among those that are in its queue of unvisited pages. Other alternatives are to use metrics that combine ordering with importance ranking discussed above, such as backlink counts or PageRank.

Let us discuss how PageRank can be used for this purpose. A slight revision is required to the PageRank formula given above. We are now modelling a random surfer: when landed on a page P, a random surfer is likely to choose one of the URLs on this page as the next one to visit with some (equal) probability d or will jump to a random page with probability $1-d$. Then the above formula for PageRank is revised as follows [Langville and Meyer, 2006]:

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

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

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

Since many web pages change over time, crawling is a continuous activity and pages need to be re-visited. Instead of restarting from scratch each time, it is preferable to selectively re-visit web pages and update the gathered information. Crawlers that follow this approach are called *incremental crawlers*. They ensure that the information in their repositories are as fresh as possible. Incremental crawlers can determine the pages that they re-visit based on the change frequency of the pages or by sampling a number of pages. *Change frequency-based* approaches use an estimate of the change frequency of a page to determine how frequently it should be re-visited [Cho and Garcia-Molina, 2000]. One might intuitively assume that pages with high change frequency should be visited more often, but this is not always true – any information extracted from a page that changes frequently is likely to become obsolete quickly, and it may be better to increase revisit interval to that page. It is also possible to develop an adaptive incremental crawler such that the crawling in one cycle is affected by the information collected in the previous cycle [Edwards et al., 2001]. *Sampling-based approaches* [Cho and Ntoulas, 2002] focus on web sites rather than individual web pages. A small number of pages from a web site are sampled to estimate how much change has happened at the site. Based on this sampling estimate, the crawler determines how frequently it should visit that site.

Some search engines specialize in searching pages belonging to a particular topic. These engines use crawlers optimized for the target topic, and are referred to as *focused crawlers*. A focused crawler ranks pages based on their relevance to the target topic, and uses them to determine which pages it should visit next. Classification techniques that are widely used in information retrieval are used in evaluating relevance. They use learning techniques to identify the topic of a given page. Learning techniques are beyond our scope, but a number of them have been developed for this purpose, such as naïve Bayes classifier [Mitchell, 1997; Chakrabarti et al., 2002], and its extensions [Passerini et al., 2001; Altingövde and Ulusoy, 2004], reinforcement learning [McCallum et al., 1999; Kaelbling et al., 1996], and others.

To achieve reasonable scale-up, crawling can be parallelized by running *parallel crawlers*. Any design for parallel crawlers must use schemes to minimize the overhead of parallelization. For instance, two crawlers running in parallel may download the same set of pages. Clearly, such overlap needs to be prevented through coordination of the crawlers' actions. One method of coordination uses a *central coordinator* to dynamically assign each crawler a set of pages to download. Another coordination scheme is to logically partition the web. Each crawler knows its partition, and there is no need for central coordination. This scheme is referred to as the *static assignment* [Cho and Garcia-Molina, 2002].

17.2.2 Indexing

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

17.2.2.1 Structure Index

The structure index is based on the graph model that we discussed in Section 17.1, with the graph representing the structure of the crawled portion of the web. The efficient storage and retrieval of these pages is important and two techniques to address these issues were discussed in Section 17.1. The structure index can be used to obtain important information about the linkage of web pages such as information regarding the *neighborhood* of a page and the siblings of a page.

17.2.2.2 Text Index

The most important and mostly used index is the *text index*. Indexes to support text-based retrieval can be implemented using any of the access methods traditionally used to search over text document collections. Examples include *suffix arrays* [Manber and Myers, 1990], *inverted files* or *inverted indexes* [Hersh, 2001], and *signature files* [Faloutsos and Christodoulakis, 1984]. Although a full treatment of all of these indexes is beyond our scope, we will discuss how inverted indexes are used in this context since these are the most popular type of text indexes.

An inverted index is a collection of inverted lists, where each list is associated with a particular word. In general, an inverted list for a given word is a list of document identifiers in which the particular word occurs [Lim et al., 2003]. If needed, the location of the word in a particular page can also be saved as part of the inverted list. This information is usually needed in proximity queries and query result ranking [Brin and Page, 1998]. Search algorithms also often make use of additional information about the occurrence of terms in a web page. For example, terms occurring in bold face (within $\langle B \rangle$ tags), in section headings (within $\langle H1 \rangle$ or $\langle H2 \rangle$ tags), or as anchor text might be weighted differently in the ranking algorithms [Arasu et al., 2001].

In addition to the inverted list, many text indexes also keep a *lexicon*, which is a list of all terms that occur in the index. The lexicon can also contain some term-level statistics that can be used by ranking algorithms [Salton, 1989].

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

1. In general, building an inverted index involves processing each page, reading all words and storing the location of each word. In the end, the inverted files are written to disk. This process, while trivial for small and static collections,

becomes hard to manage when dealing with a vast and non-static collection like the web.

2. The rapid change of the web poses the second challenge for maintaining the “freshness” of the index. Although we argued in the previous section that incremental crawlers should be deployed to ensure freshness, it has also been argued that periodic index rebuilding is still necessary because most incremental update techniques do not perform well when dealing with the large changes often observed between successive crawls [Melnik et al., 2001].
3. Storage formats of inverted indexes must be carefully designed. There is a tradeoff between a performance gain through a compressed index that allows portions of the index to be cached in memory, and the overhead of decompression at query time. Achieving the right balance becomes a major concern when dealing with web-scale collections.

Addressing these challenges and developing a highly scalable text index can be achieved by distributing the index by either building a *local inverted index* at each machine where the search engine runs or building a *global inverted index* that is then shared [Ribeiro-Neto and Barbosa, 1998]. We don’t discuss these further, as the issues are similar to the distributed data and directory management issues we have already covered in previous chapters.

17.2.3 Ranking and Link Analysis

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

Link-based algorithms can be used to rank a collection of pages. To repeat what we discussed earlier, the intuition is that if a page P_j contains a link to page P_i, then it is likely that the authors of page P_j think that page P_i is of good quality. Thus, a page that has a large number of incoming links is expected to have good quality, and hence the number of incoming links to a page can be used as a ranking criteria. This intuition is the basis of ranking algorithms, but, of course, the each specific algorithm implements this intuition in a different and sophisticated way. We already discussed the PageRank algorithm earlier. We will discuss an alternative algorithm called HITS to highlight different ways of approaching the issue [Kleinberg, 1999].

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

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

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

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

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

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

17.2.4 Evaluation of Keyword Search

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

Category search addresses one of the problems of using keyword search, namely the lack of a global view of the web. Category search is also known as web directory, catalogs, yellow pages, and subject directories. There are a number of public web directories available: dmoz (http://dmoz.org/), LookSmart (http://www.looksmart.com/), and Yahoo (http://www.yahoo.com/). The web directory is a hierarchical taxonomy that classifies human knowledge [Baeza-Yates and Ribeiro-Neto, 1999]. Although, the taxonomy is typically displayed as a tree, it is actually a directed acyclic graph since some categories are cross referenced,.

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

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

17.3 Web Querying

Declarative querying and efficient execution of queries has been a major focus of database technology. It would be beneficial if the database techniques can be applied to the web. In this way, accessing the web can be treated, to a certain extent, similar to accessing a large database.

There are difficulties in carrying over traditional database querying concepts to web data. Perhaps the most important difficulty is that database querying assumes the existence of a strict schema. As noted above, it is hard to argue that there is a schema for web data similar to databases[2]. At best, the web data are *semistructured* – data may have some structure, but this may not be as rigid, regular, or complete as that of databases, so that different instances of the data may be similar but not identical (there may be missing or additional attributes or differences in structure). There are, obviously, inherent difficulties in querying schema-less data.

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

A third major difficulty is that there is no commonly accepted language, similar to SQL, for querying web data. As we noted in the previous section, keyword search has a very simple language, but this is not sufficient for richer querying of web data. Some consensus on the basic constructs of such a language has emerged (e.g., path expressions), but there is no standard language. However, a standardized language

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

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

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

17.3.1 Semistructured Data Approach

One way to approach querying the web data is to treat it as a collection of semistructured data. Then, models and languages that have been developed for this purpose can be used to query the data. Semistructured data models and languages were not originally developed to deal with web data; rather they addressed the requirements of growing data collections that did not have as strict a schema as their relational counterparts. However, since these characteristics are also common to web data, later studies explored their applicability in this domain. We demonstrate this approach using a particular model (OEM) and a language (Lorel), but other approaches such as UnQL [Buneman et al., 1996] are similar.

OEM (Object Exchange Model) [Papakonstantinou et al., 1995] is a self-describing semistructured data model. Self-describing means that each object specifies the schema that it follows.

An OEM object is defined as a four-tuple ⟨`label, type, value, oid`⟩, where `label` is a character string describing what the object represents, `type` specifies the type of the object's value, `value` is obvious, and `oid` is the object identifier that distinguishes it from other objects. The type of an object can be `atomic`, in which case the object is called an *atomic object*, or `complex`, in which case the object is called a *complex object*. An atomic object contains a primitive value such as an integer, a real, or a string, while a complex object contains a set of other objects, which can themselves be atomic or complex. The value of a complex object is a set of oids. One would immediately recognize the similarity between OEM object definition and the object models that we discussed in Chapter 15.

Example 17.1. Let us consider a bibliographic database that consists of a number of documents. A snapshot of an OEM representation of such a database is given in Figure 17.5. Each line shows one OEM object and the indentation is provided to simplify the display of the object structure. For example, the second line `<doc, complex, &o3, &o6, &o7, &o20, &o21, &o2>` defines an object whose label is `doc`, type is `complex`, oid is `&o2`, and whose value consists of objects whose oids are `&o3, &o6, &o7, &o20`, and `&o21`.

This database contains three documents (`&o2, &o22, &034`); the first and third are books and the second is an article. There are commonalities among the two books (and even the article), but there are differences as well. For example, the first book (`&o2`) has the price information that the second one (`&o34`) does not have, while the second one has ISBN and publisher information that the first does not have. The object-oriented structure of the database is obvious – complex objects consist of

```
<bib, complex, {&o2, &o22, &034}, &o1>
   <doc, complex, {&o3, &o6, &o7, &o20, &o22}, &o2>
      <authors, complex, {&o4, &o5}, &o3>
         <author, string, "M. Tamer Ozsu", &o4>
         <author, string, "Patrick Valduriez", &o5>
      <title, string, "Principles of Distributed ...", &o6>
      <chapters, complex, {&o8, &o11, &o14, &o17}, &o7>
         <chapter, complex, {&o9, &o10}, &o8>
            <heading, string, "...", &o9>
            <body, string, "...", &o10>
            ...
         <chapter, complex, {&o18, &o19}, &17>
            <heading, string, "...", &o18>
            <body, string, "...", &o19>
      <what, string, "Book", &o20>
      <price, float, 98.50, &o21>
   <doc, complex, {&o23, &o25, &o26, &o27, &o28}, &o22>
      <authors, complex, {&o24, &o4}, &o23>
         <author, string, "Yingying Tao", &o24>
      <title, string, "Mining data streams ...", &o25>
      <venue, string, "CIKM", &o26>
      <year, integer, 2009, &o27>
      <sections, complex, {&o29, &o30, &o31, &o32, &o33}, &28>
         <section, string, "...", &o29>
           ...
         <section, string, "...", &o33>
   <doc, complex, {&o16,&o17,&o7,&o18,&o19,&o20,&o21},&o34>
      <author, string, "Anthony Bonato", &o35>
      <title, string, "A Course on the Web Graph", &o36>
      <what, string, "Book", &o20>
      <ISBN, string, "TK5105.888.B667", &o37>
      <chapters, complex, {&o39, &o42, &o45}, &o38>
         <chapter, complex, {&o40, &o41}, &o39>
            <heading, string, "...", &o40>
            <body, string, "...", &o41>
         <chapter, complex, {&o43, &o44}, &o42>
            <heading, string, "...", &o43>
            <body, string, "...", &o44>
         <chapter, complex, {&o46, &o47}, &45>
            <heading, string, "...", &o46>
            <body, string, "...", &o47>
      <publisher, string, "AMS", &o48>
```

Fig. 17.5 An Example OEM Specification

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

As noted earlier, OEM data are self-describing, where each object identifies itself through its type and its label. It is easy to see that the OEM data can be represented as a node-labelled graph where the nodes correspond to each OEM object and the edges correspond to the subobject relationship. The label of a node is the oid and the label of the object corresponding to that node. However, it is quite common in literature to model the data as an edge-labelled graph: if object o_j is a subobject of object o_i, then o_j's label is assigned to the edge connecting o_i to o_j, and the oids are omitted as node labels. In Example 17.2, we use a node and edge-labelled representation that shows oids as node labels and assigns edge labels as described above.

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

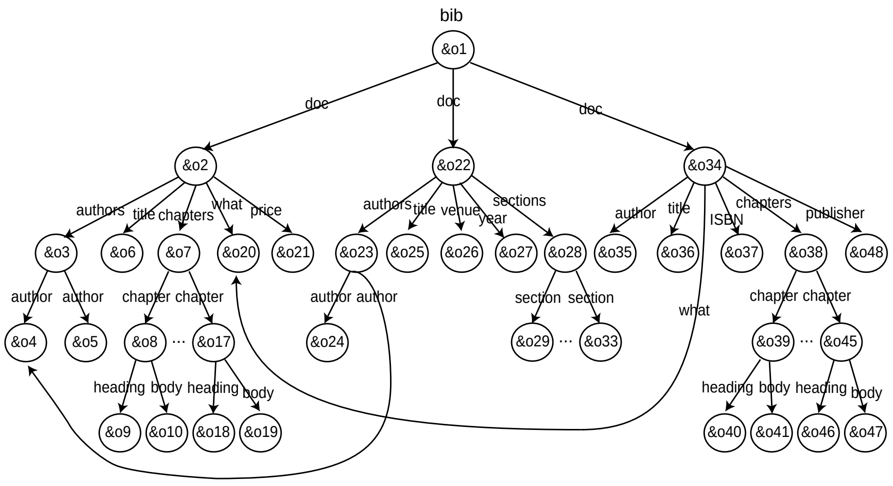


Fig. 17.6 The corresponding OEM graph for the OEM database of Example 17.1

The semistructured approach fits reasonably well for modelling web data that can be represented as a graph. Furthermore, it accepts that data may have some structure, but this may not be as rigid, regular, or complete as that of traditional databases. The users do not need to be aware of the complete structure when they query the data. Therefore, expressing a query should not require full knowledge of the structure. These graph representations of data at each data source are generated by wrappers that we discussed in Chapter 9.

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

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

The fundamental construct in forming Lorel queries is, therefore, a *path expression*. We discussed path expressions as they appear in object database systems in Section 15.6.2.2, but we give the definition here as it applies to Lore. In its simplest form, a path expression in Lorel is a sequence of labels starting with an object name or a variable denoting an object. For example `bib.doc.title` is a path expression whose interpretation is to start at bib and follow the edge labelled doc and then follow the edge labelled title. Note that there are three paths in Figure 17.6 that would satisfy this expression: (i) `&o1.doc:&o2.title:&o6`, (ii) `&o1.doc:&o22.title:&o25`, and (iii) `&o1.doc:&o34.title:&o36`. Each of these are called a *data path*. In Lorel, path expressions can be more complex regular expressions such that what follows the object name or variable is not only a label, but more general expressions that can be constructed using conjunction, disjunction (|), iteration (? to mean 0 or 1 occurrences, + to mean 1 or more, and $*$ to mean 0 or more), and wildcards (#).

Example 17.3. The following are examples of acceptable path expressions in Lorel:

(a) `bib.doc(.authors)?.author` : start from `bib`, follow `doc` edge and the `author` edge with an optional `authors` edge in between.

(b) `bib.doc.#.author` : start from `bib`, follow `doc` edge, then an arbitrary number of edges with unspecified labels (using the wildcard #), and follow the `author` edge.

(c) `bib.doc.%price` : start from `bib`, follow `doc` edge, then an edge whose label has the string "price" preceded by some characters.

♦

Example 17.4. The following are example Lorel queries that use some of the path expressions given in Example 17.3:

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

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

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

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

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

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

♦

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

Finally, the graph structure can get quite complicated, making it difficult to query. Although Lorel provides a number of features (such as wildcards) to make querying easier, the examples above indicate that a user still needs to know the general structure of the semistructured data. The OEM graphs for large databases can become quite complicated, and it is hard for users to form the path expressions. The issue, then, is how to "summarize" the graph so that there might be a reasonably small schema-like description that might aid querying. For this purpose, a construct called a DataGuide [Goldman and Widom, 1997] has been proposed. A DataGuide is a graph where each path in the corresponding OEM graph occurs only once. It is dynamic in that as the OEM graph changes, the corresponding DataGuide is updated. Thus, it provides concise and accurate structural summaries of semistructured databases and can be used as a light-weight schema, which is useful for browsing the database structure, formulating queries, storing statistical information, and enabling query optimization.

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

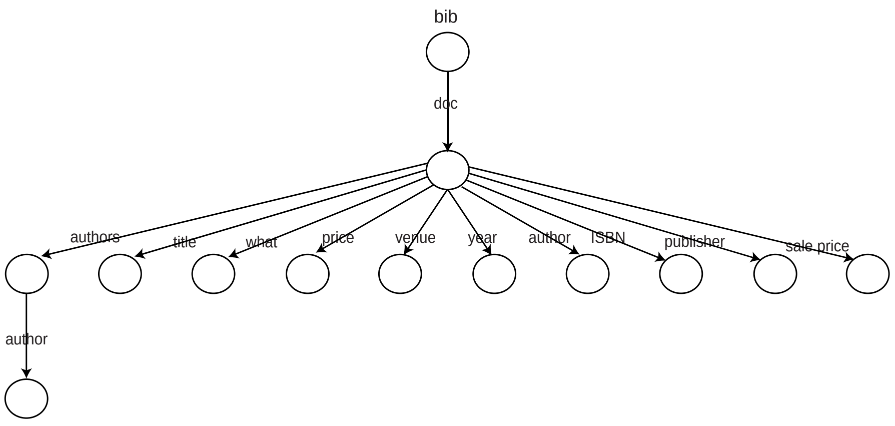


Fig. 17.7 The DataGuide corresponding to the OEM graph of Example 17.2

17.3.2 Web Query Language Approach

The approaches in this category are aimed to directly address the characteristics of web data, particularly focusing on handling *links* properly. Their starting point is to overcome the shortcomings of keyword search by providing proper abstractions for capturing the content structure of documents (as in semistructured data approaches) as well as the external links. They combine the content-based queries (e.g., keyword expressions) and structure-based queries (e.g., path expressions).

A number of languages have been proposed specifically to deal with web data, and these can be categorized as first-generation and second generation [Florescu et al., 1998]. The first generation languages model the web as interconnected collection of *atomic* objects. Consequently, these languages can express queries that search the link structure among web objects and their textual content, but they cannot express queries that exploit the document structure of these web objects. The second generation languages model the web as a linked collection of *structured* objects, allowing them to express queries that exploit the document structure similar to semistructured languages. First generation approaches include WebSQL [Mendelzon et al., 1997], W3QL [Konopnicki and Shmueli, 1995], and WebLog [Lakshmanan et al., 1996], while second generation approaches include WebOQL [Arocena and Mendelzon, 1998], and StruQL [Fernandez et al., 1997]. We will demonstrate the general ideas by considering one first generation language (WebSQL) and one second generation language (WebOQL).

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

DOCUMENT(URL, TITLE, TEXT, TYPE, LENGTH, MODIF)

ANCHOR(BASE, HREF, LABEL)

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

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

(a) *interior link* that exists within the same document (#>)

(b) *local link* that is between documents on the same server (->)

(c) *global link* that refers to a document on another server (=>)

(d) *null path* (=)

These link types form the alphabet of the path expressions. Using them, and the usual constructors of regular expressions, different paths can be specified as in Example 17.6.

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

(a) -> | =>: a path of length one, either local or global

(b) ->*: local path of any length

(c) =>->*: as above, but in other servers

(d) (-> |=>)*: the reachable portion of the web

♦

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

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

where `domain-condition` can be either a path expression, or can specify a text search using `MENTIONS`, or can specify that an attribute (in the `SELECT` clause) is equal to a web object. Of course, following each relation specification, there could be a variable ranging over the relation – this is standard SQL. The following example queries (taken from [Mendelzon et al., 1997] with minor modifications) demonstrate the features of WebSQL.

Example 17.7. Following are some examples of WebSQL:

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

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

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

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

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

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

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

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

♦

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

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

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

Example 17.8. Let us revisit Example 17.1 and assume that instead of modelling the documents in a bibliography, it models the collection of documents about data management over the web. A possible (partial) hypertree for this example is given in Figure 17.8. Note that we have made one revision to facilitate some of the queries to be discussed later: we added an abstract to each document.

In Figure 17.8, the documents are first grouped along a number of topics as indicated in the records attached to the edges from the root. In this representation, the internal links are shown as solid edges and external links as dashed edges. Recall that in OEM (Figure 17.6), the edges represent both attributes (e.g., author) and document structure (e.g., chapter). In the WebOQL model, the attributes are captured in the records that are associated with each edge, while the (internal) edges represent the document structure. ♦

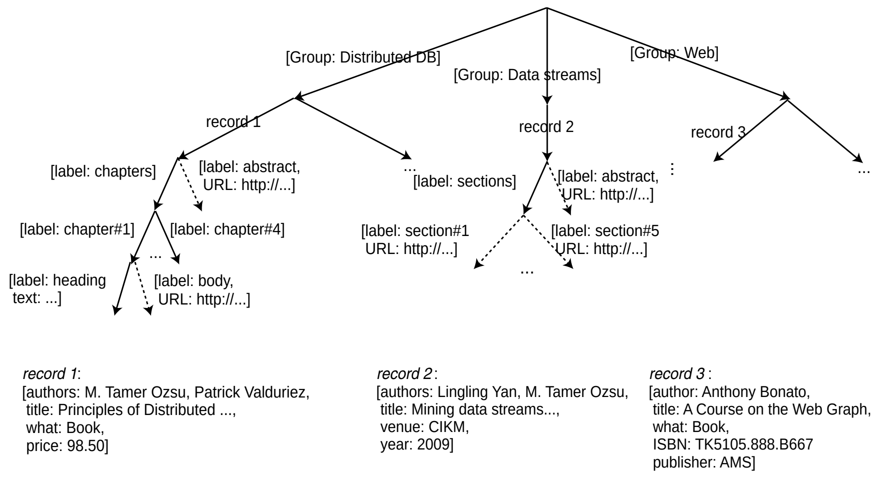

[Group: Distributed DB]
[Group: Data streams]
[Group: Web]
record 1
record 2
record 3
[label: chapters]
[label: abstract, URL: http://...]
[label: sections]
[label: abstract, URL: http://...]
[label: chapter#1]
[label: chapter#4]
[label: section#1 URL: http://...]
[label: section#5 URL: http://...]
[label: heading text: ...]
[label: body, URL: http://...]
record 1: [authors: M. Tamer Ozsu, Patrick Valduriez, title: Principles of Distributed ..., what: Book, price: 98.50]
record 2: [authors: Lingling Yan, M. Tamer Ozsu, title: Mining data streams..., venue: CIKM, year: 2009]
record 3: [author: Anthony Bonato, title: A Course on the Web Graph, what: Book, ISBN: TK5105.888.B667 publisher: AMS]

Fig. 17.8 The hypertree example

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

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

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

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

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

Concatenate: combines two trees (denoted +).

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

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

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

In addition to these, WebOQL introduces a string pattern matching operator (denoted $\sim$) whose left argument is a string and right argument is a string pattern.