

A major problem with a primary index—as with any ordered file—is insertion and deletion of records. With a primary index, the problem is compounded because if we attempt to insert a record in its correct position in the data file, we must not only move records to make space for the new record but also change some index entries, since moving records will change the *anchor records* of some blocks. Using an unordered overflow file, as discussed in Section 16.7, can reduce this problem. Another possibility is to use a linked list of overflow records for each block in the data file. This is similar to the method of dealing with overflow records described with hashing in Section 16.8.2. Records within each block and its overflow linked list can be sorted to improve retrieval time. Record deletion is handled using deletion markers.

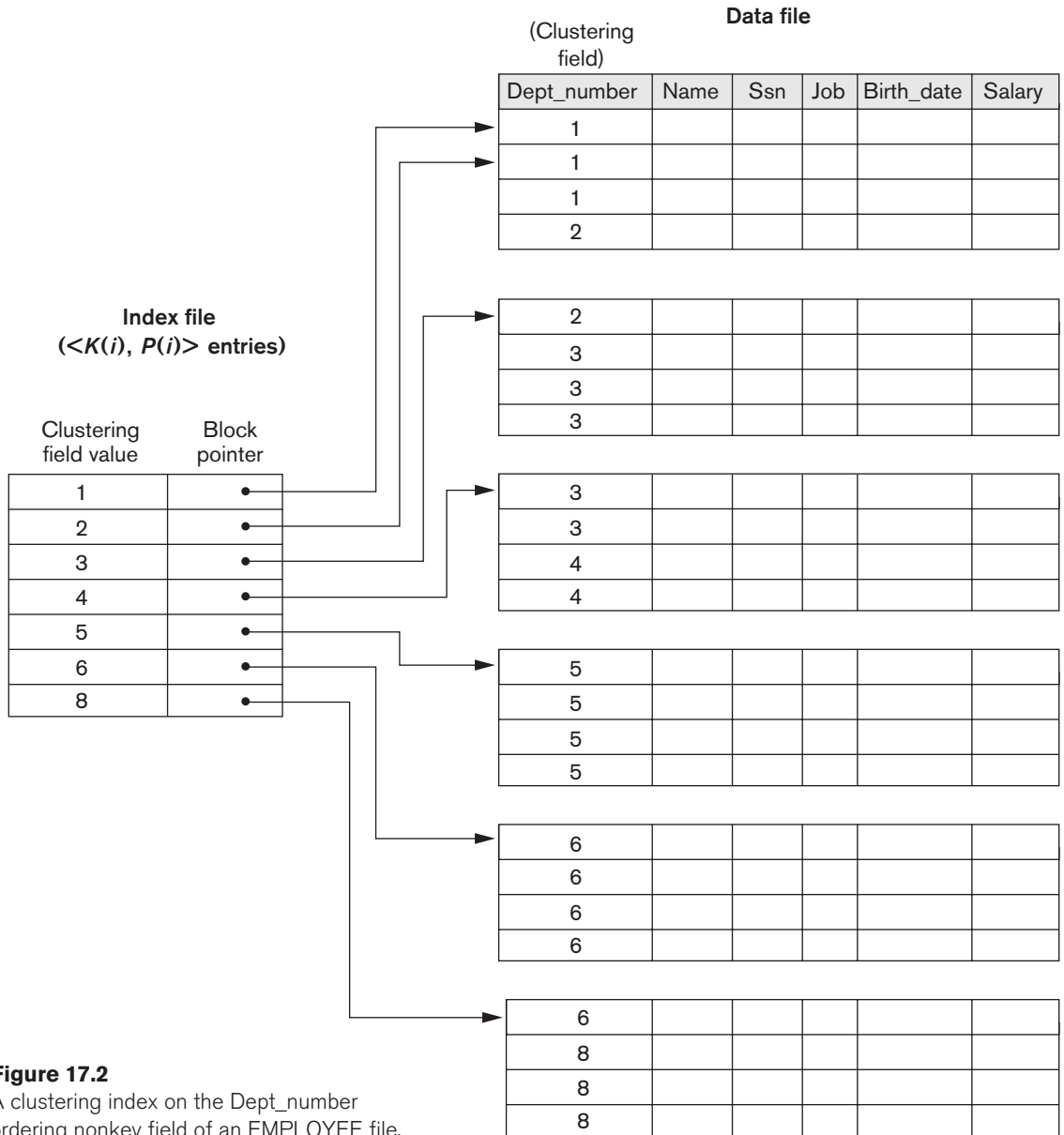
### 17.1.2 Clustering Indexes

If file records are physically ordered on a nonkey field—which *does not* have a distinct value for each record—that field is called the **clustering field** and the data file is called a **clustered file**. We can create a different type of index, called a **clustering index**, to speed up retrieval of all the records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a *distinct value* for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, and it contains the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. Figure 17.2 shows an example. Notice that record insertion and deletion still cause problems because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for *each value* of the clustering field; all records with that value are placed in the block (or block cluster). This makes insertion and deletion relatively straightforward. Figure 17.3 shows this scheme.

A clustering index is another example of a *nondense* index because it has an entry for every *distinct value* of the indexing field, which is a nonkey by definition and hence has duplicate values rather than a unique value for every record in the file.

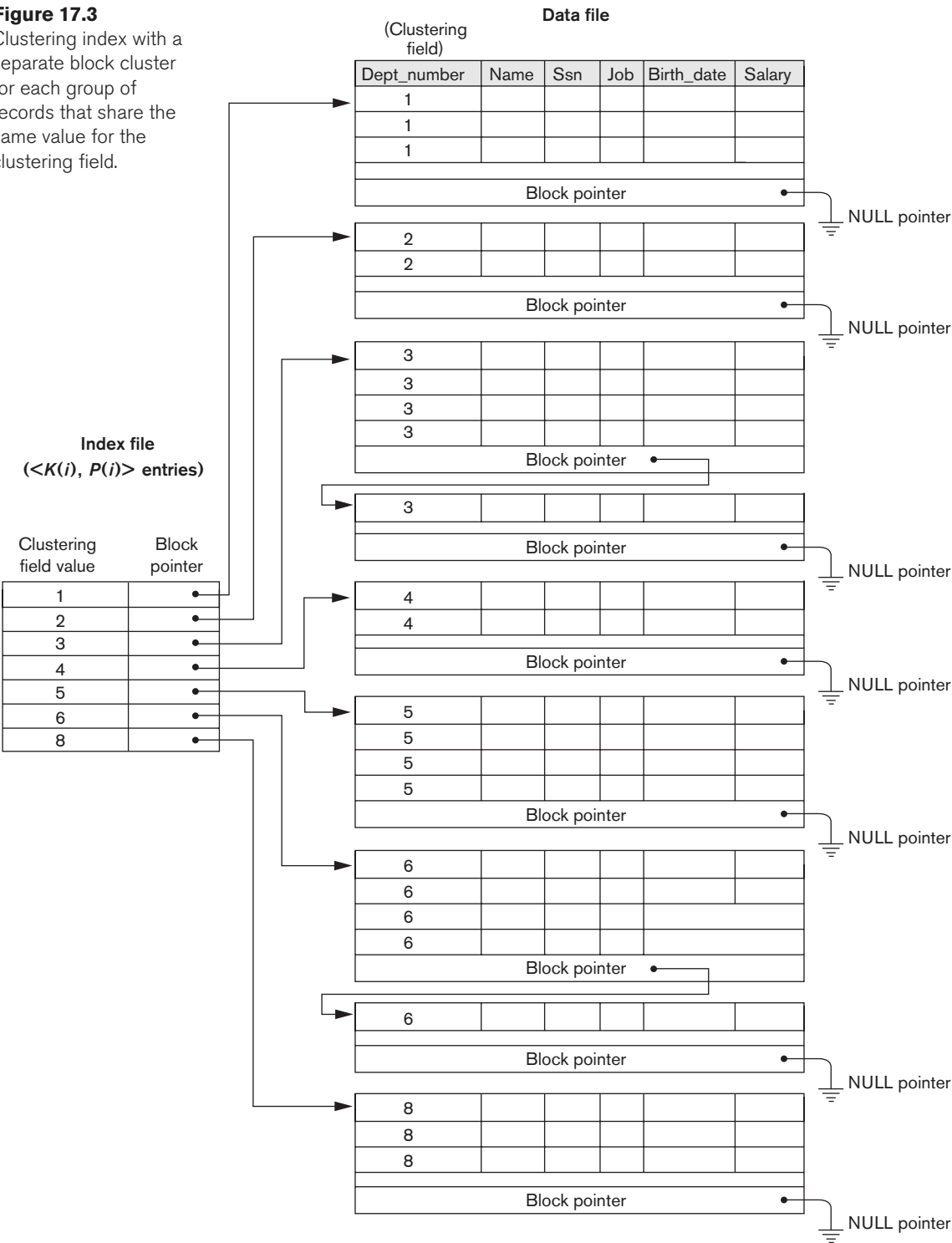
**Example 2.** Suppose that we consider the same ordered file with  $r = 300,000$  records stored on a disk with block size  $B = 4,096$  bytes. Imagine that it is ordered by the attribute Zipcode and there are 1,000 zip codes in the file (with an average 300 records per zip code, assuming even distribution across zip codes.) The index in this case has 1,000 index entries of 11 bytes each (5-byte Zipcode and 6-byte block pointer) with a blocking factor  $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4,096/11) \rfloor = 372$  index entries per block. The number of index blocks is hence  $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (1,000/372) \rceil = 3$  blocks. To perform a binary search on the index file would need  $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 3) \rceil = 2$  block accesses. Again, this index would typically be loaded in main memory (occupies 11,000 or 11 Kbytes) and takes negligible time to search in memory. One block access to the data file would lead to the first record with a given zip code.

**Figure 17.2**

A clustering index on the Dept\_number ordering nonkey field of an EMPLOYEE file.

There is some similarity between Figures 17.1, 17.2, and 17.3 and Figures 16.11 and 16.12. An index is somewhat similar to dynamic hashing (described in Section 16.8.3) and to the directory structures used for extendible hashing. Both are searched to find a pointer to the data block containing the desired record. A main difference is that an index search uses the values of the search field itself, whereas a hash directory search uses the binary hash value that is calculated by applying the hash function to the search field.

**Figure 17.3**  
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



### 17.1.3 Secondary Indexes

A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed. The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a nonkey field with duplicate values. The index is again an ordered file with two fields. The first field is of the same data type as some *nonordering field* of the data file that is an **indexing field**. The second field is either a *block* pointer or a *record* pointer. Many secondary indexes (and hence, indexing fields) can be created for the same file—each represents an additional means of accessing that file based on some specific field.

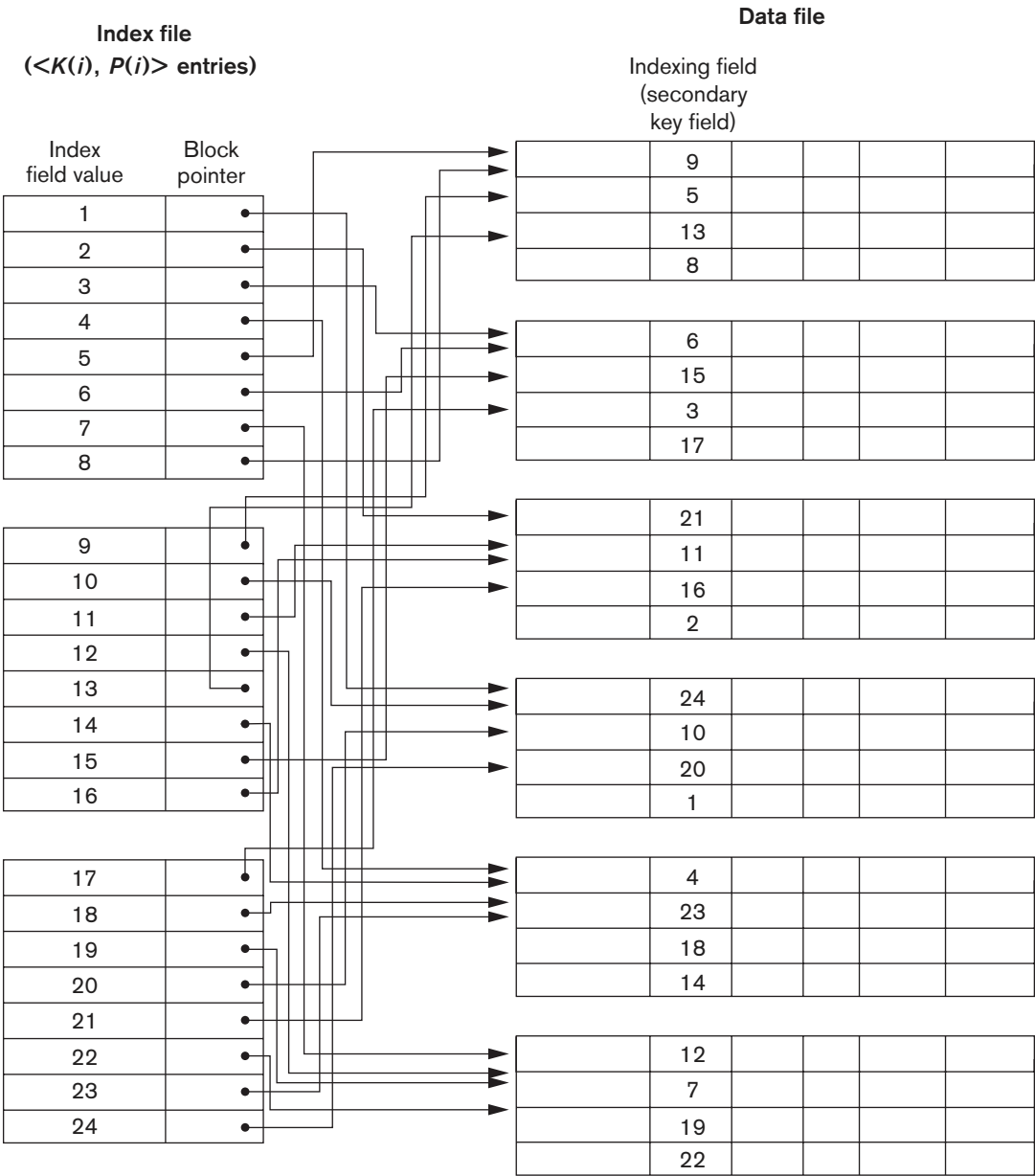
First we consider a secondary index access structure on a key (unique) field that has a *distinct value* for every record. Such a field is sometimes called a **secondary key**; in the relational model, this would correspond to any UNIQUE key attribute or to the primary key attribute of a table. In this case there is one index entry for *each record* in the data file, which contains the value of the field for the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is **dense**.

Again we refer to the two field values of index entry  $i$  as  $\langle K(i), P(i) \rangle$ . The entries are **ordered** by value of  $K(i)$ , so we can perform a binary search. Because the records of the data file are *not* physically ordered by values of the secondary key field, we *cannot* use block anchors. That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index. Figure 17.4 illustrates a secondary index in which the pointers  $P(i)$  in the index entries are *block pointers*, not record pointers. Once the appropriate disk block is transferred to a main memory buffer, a search for the desired record within the block can be carried out.

A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, the *improvement* in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a *linear search* on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main file, even if the index did not exist. Example 3 illustrates the improvement in number of blocks accessed.

**Example 3.** Consider the file of Example 1 with  $r = 300,000$  fixed-length records of size  $R = 100$  bytes stored on a disk with block size  $B = 4,096$  bytes. The file has  $b = 7,500$  blocks, as calculated in Example 1. Suppose we want to search for a record with a specific value for the secondary key—a nonordering key field of the file that is  $V = 9$  bytes long. Without the secondary index, to do a linear search on the file would require  $b/2 = 7,500/2 = 3,750$  block accesses on the average. Suppose that we construct a secondary index on that *nonordering key* field of the file. As in Example 1, a block pointer is  $P = 6$  bytes long, so each index entry is  $R_i = (9 + 6) = 15$  bytes, and the blocking factor for the index is  $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4,096/15) \rfloor = 273$  index entries per block. In a dense secondary index such as this, the total number of index entries  $r_i$  is equal to the *number of records* in the data file, which is 300,000. The number of blocks needed for the index is hence  $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (300,000/273) \rceil = 1,099$  blocks.

**Figure 17.4**  
A dense secondary index (with block pointers) on a nonordering key field of a file.

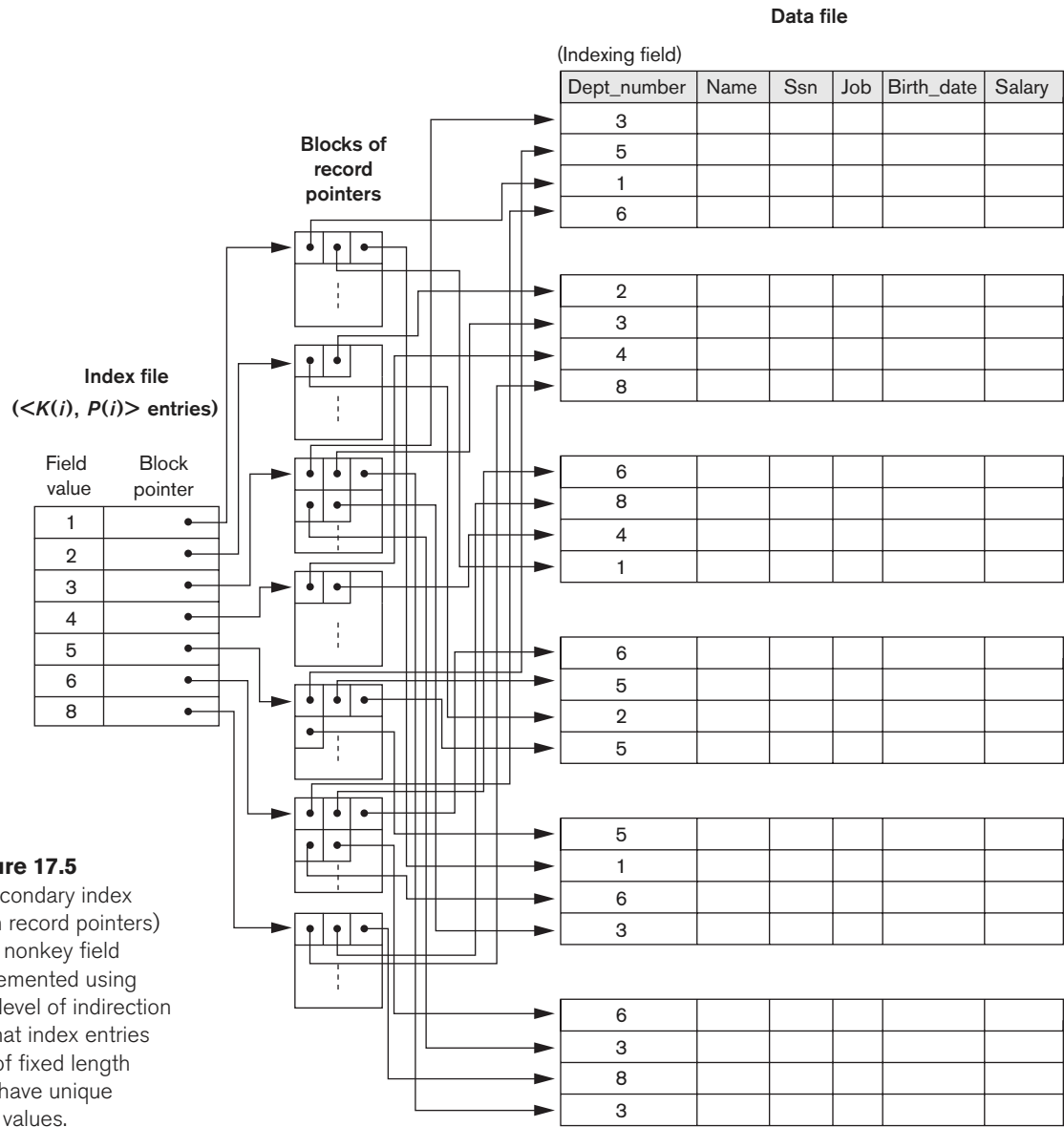


A binary search on this secondary index needs  $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 1,099) \rceil = 11$  block accesses. To search for a record using the index, we need an additional block access to the data file for a total of  $11 + 1 = 12$  block accesses—a vast improvement over the 3,750 block accesses needed on the average for a linear search, but slightly worse than the 6 block accesses required for the primary index. This difference arose because the primary index was nondense and hence shorter, with only 28 blocks in length as opposed to the 1,099 blocks dense index here.

We can also create a secondary index on a *nonkey, nonordering field* of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:

- Option 1 is to include duplicate index entries with the same  $K(i)$  value—one for each record. This would be a dense index.
- Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer. We keep a list of pointers  $\langle P(i, 1), \dots, P(i, k) \rangle$  in the index entry for  $K(i)$ —one pointer to each block that contains a record whose indexing field value equals  $K(i)$ . In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately to account for a variable number of index entries per index key value.
- Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each *index field value*, but to create *an extra level of indirection* to handle the multiple pointers. In this nondense scheme, the pointer  $P(i)$  in index entry  $\langle K(i), P(i) \rangle$  points to a disk block, which contains a *set of record pointers*; each record pointer in that disk block points to one of the data file records with value  $K(i)$  for the indexing field. If some value  $K(i)$  occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used. This technique is illustrated in Figure 17.5. Retrieval via the index requires one or more additional block accesses because of the extra level, but the algorithms for searching the index and (more importantly) for inserting of new records in the data file are straightforward. The binary search algorithm is directly applicable to the index file since it is ordered. For range retrievals such as retrieving records where  $V_1 \leq K \leq V_2$ , block pointers may be used in the pool of pointers for each value instead of the record pointers. Then a union operation can be used on the pools of block pointers corresponding to the entries from  $V_1$  to  $V_2$  in the index to eliminate duplicates and the resulting blocks can be accessed. In addition, retrievals on complex selection conditions may be handled by referring to the record pointers from multiple non-key secondary indexes, without having to retrieve many unnecessary records from the data file (see Exercise 17.24).

Notice that a secondary index provides a **logical ordering** on the records by the indexing field. If we access the records in order of the entries in the secondary index, we get them in order of the indexing field. The primary and clustering indexes assume that the field used for **physical ordering** of records in the file is the same as the indexing field.



**Figure 17.5**  
A secondary index  
(with record pointers)  
on a nonkey field  
implemented using  
one level of indirection  
so that index entries  
are of fixed length  
and have unique  
field values.

**17.1.4 Summary**

To conclude this section, we summarize the discussion of index types in two tables. Table 17.1 shows the index field characteristics of each type of ordered single-level index discussed—primary, clustering, and secondary. Table 17.2 summarizes the properties of each type of index by comparing the number of index entries and specifying which indexes are dense and which use block anchors of the data file.

**Table 17.1** Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

**Table 17.2** Properties of Index Types

Type of Index	Number of (First-Level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no <sup>a</sup>
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records <sup>b</sup> or number of distinct index field values <sup>c</sup>	Dense or Nondense	No

<sup>a</sup>Yes if every distinct value of the ordering field starts a new block; no otherwise.

<sup>b</sup>For option 1.

<sup>c</sup>For options 2 and 3.

## 17.2 Multilevel Indexes

The indexing schemes we have described thus far involve an ordered index file. A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file having a specific index field value. A binary search requires approximately  $(\log_2 b_i)$  block accesses for an index with  $b_i$  blocks because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2. This is why we take the log function to the base 2. The idea behind a **multilevel index** is to reduce the part of the index that we continue to search by  $bfr_i$ , the blocking factor for the index, which is larger than 2. Hence, the search space is reduced much faster. The value  $bfr_i$  is called the **fan-out** of the multilevel index, and we will refer to it by the symbol **fo**. Whereas we divide the *record search space* into two halves at each step during a binary search, we divide it  $n$ -ways (where  $n$  = the fan-out) at each search step using the multilevel index. Searching a multilevel index requires approximately  $(\log_{fo} b_i)$  block accesses, which is a substantially smaller number than for a binary search if the fan-out is larger than 2. In most cases, the fan-out is much larger than 2. Given a blocksize of 4,096, which is most common in today's DBMSs, the fan-out depends on how many (key + block pointer) entries fit within a block. With a 4-byte block pointer (which would accommodate  $2^{32} - 1 = 4.2 \times 10^9$  blocks) and a 9-byte key such as SSN, the fan-out comes to 315.

A multilevel index considers the index file, which we will now refer to as the **first** (or **base**) **level** of a multilevel index, as an *ordered file* with a *distinct value* for each



$K(i)$ . Therefore, by considering the first-level index file as a sorted data file, we can create a primary index for the first level; this index to the first level is called the **second level** of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for *each block* of the first level. The blocking factor  $bfr_i$  for the second level—and for all subsequent levels—is the same as that for the first-level index because all index entries are the same size; each has one field value and one block address. If the first level has  $r_1$  entries, and the blocking factor—which is also the fan-out—for the index is  $bfr_i = fo$ , then the first level needs  $\lceil (r_1/fo) \rceil$  blocks, which is therefore the number of entries  $r_2$  needed at the second level of the index.

We can repeat this process for the second level. The **third level**, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is  $r_3 = \lceil (r_2/fo) \rceil$ . Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block. We can repeat the preceding process until all the entries of some index level  $t$  fit in a single block. This block at the  $t$ th level is called the **top** index level.<sup>6</sup> Each level reduces the number of entries at the previous level by a factor of  $fo$ —the index fan-out—so we can use the formula  $1 \leq (r_1/((fo)^t))$  to calculate  $t$ . Hence, a multilevel index with  $r_1$  first-level entries will have approximately  $t$  levels, where  $t = \lceil (\log_{fo}(r_1)) \rceil$ . When searching the index, a single disk block is retrieved at each level. Hence,  $t$  disk blocks are accessed for an index search, where  $t$  is the *number of index levels*.

The multilevel scheme described here can be used on any type of index—whether it is primary, clustering, or secondary—as long as the first-level index has *distinct values for  $K(i)$  and fixed-length entries*. Figure 17.6 shows a multilevel index built over a primary index. Example 3 illustrates the improvement in number of blocks accessed when a multilevel index is used to search for a record.

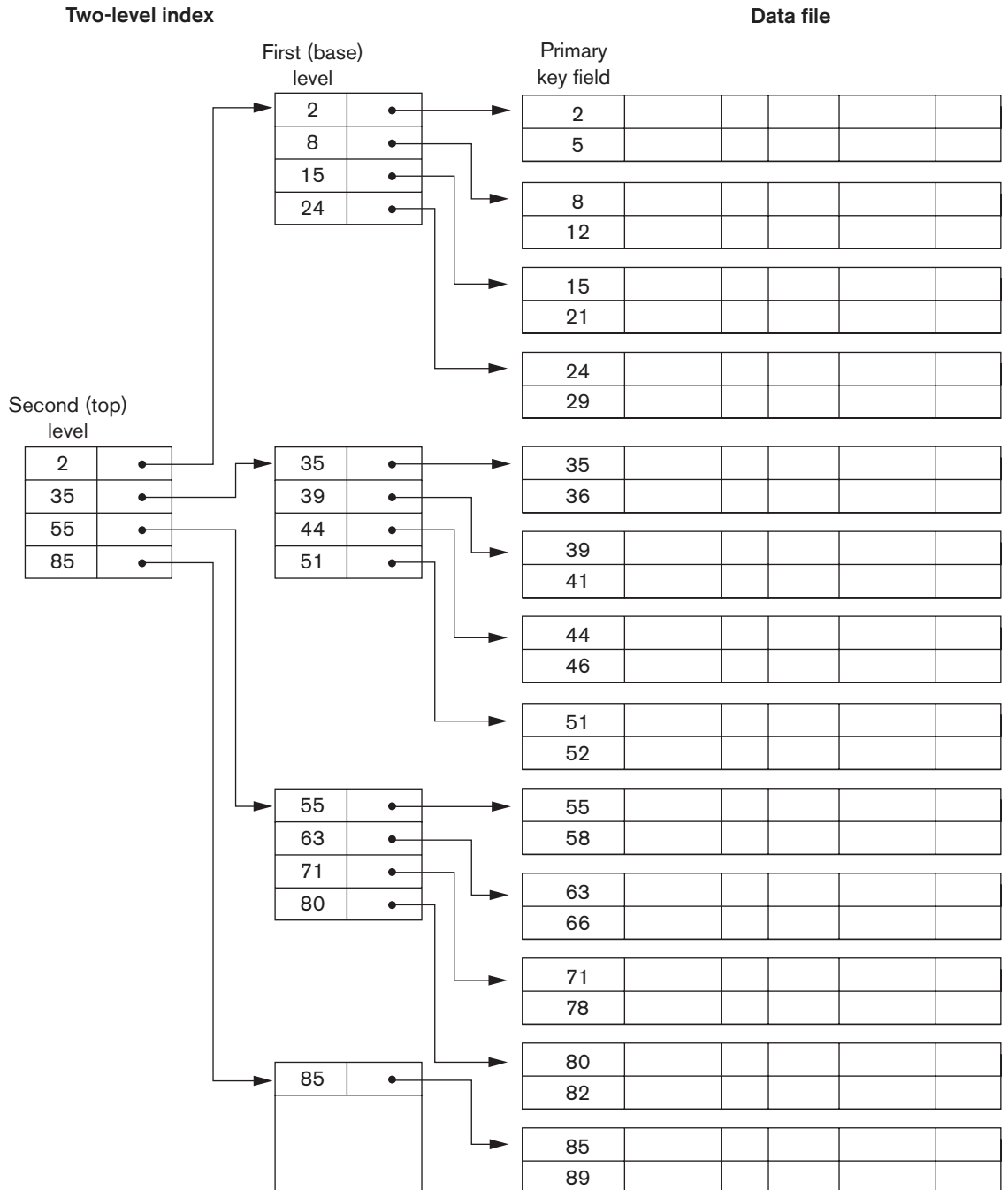
**Example 4.** Suppose that the dense secondary index of Example 3 is converted into a multilevel index. We calculated the index blocking factor  $bfr_i = 273$  index entries per block, which is also the fan-out  $fo$  for the multilevel index; the number of first-level blocks  $b_1 = 1,099$  blocks was also calculated. The number of second-level blocks will be  $b_2 = \lceil (b_1/fo) \rceil = \lceil (1,099/273) \rceil = 5$  blocks, and the number of third-level blocks will be  $b_3 = \lceil (b_2/fo) \rceil = \lceil (5/273) \rceil = 1$  block. Hence, the third level is the top level of the index, and  $t = 3$ . To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need  $t + 1 = 3 + 1 = 4$  block accesses. Compare this to Example 3, where 12 block accesses were needed when a single-level index and binary search were used.

Notice that we could also have a multilevel primary index, which would be non-dense. Exercise 17.18(c) illustrates this case, where we *must* access the data block from the file before we can determine whether the record being searched for is in the file. For a dense index, this can be determined by accessing the first index level

<sup>6</sup>The numbering scheme for index levels used here is the reverse of the way levels are commonly defined for tree data structures. In tree data structures,  $t$  is referred to as level 0 (zero),  $t - 1$  is level 1, and so on.

**Figure 17.6**

A two-level primary index resembling ISAM (indexed sequential access method) organization.



(without having to access a data block), since there is an index entry for *every* record in the file.

A common file organization used in business data processing is an ordered file with a multilevel primary index on its ordering key field. Such an organization is called an **indexed sequential file** and was used in a large number of early IBM systems. IBM's **ISAM** organization incorporates a two-level index that is closely related to the organization of the disk in terms of cylinders and tracks (see Section 16.2.1). The first level is a cylinder index, which has the key value of an anchor record for each cylinder of a disk pack occupied by the file and a pointer to the track index for the cylinder. The track index has the key value of an anchor record for each track in the cylinder and a pointer to the track. The track can then be searched sequentially for the desired record or block. Insertion is handled by some form of overflow file that is merged periodically with the data file. The index is re-created during file reorganization.

Algorithm 17.1 outlines the search procedure for a record in a data file that uses a nondense multilevel primary index with  $t$  levels. We refer to entry  $i$  at level  $j$  of the index as  $\langle K_j(i), P_j(i) \rangle$ , and we search for a record whose primary key value is  $K$ . We assume that any overflow records are ignored. If the record is in the file, there must be some entry at level 1 with  $K_1(i) \leq K < K_1(i + 1)$  and the record will be in the block of the data file whose address is  $P_1(i)$ . Exercise 17.23 discusses modifying the search algorithm for other types of indexes.

**Algorithm 17.1.** Searching a Nondense Multilevel Primary Index with  $t$  Levels

(\*We assume the index entry to be a block anchor that is the first key per block\*)

$p \leftarrow$  address of top-level block of index;

for  $j \leftarrow t$  step  $-1$  to  $1$  do

begin

read the index block (at  $j$ th index level) whose address is  $p$ ;

search block  $p$  for entry  $i$  such that  $K_j(i) \leq K < K_j(i + 1)$

(\* if  $K_j(i)$

is the last entry in the block, it is sufficient to satisfy  $K_j(i) \leq K$  \*);

$p \leftarrow P_j(i)$  (\* picks appropriate pointer at  $j$ th index level \*)

end;

read the data file block whose address is  $p$ ;

search block  $p$  for record with key  $= K$ ;

As we have seen, a multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value. We are still faced with the problems of dealing with index insertions and deletions, because all index levels are *physically ordered files*. To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index called a **dynamic multilevel index** that leaves some space in each of its blocks for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks when the data file grows and shrinks. It is often implemented by using data structures called B-trees and B<sup>+</sup>-trees, which we describe in the next section.

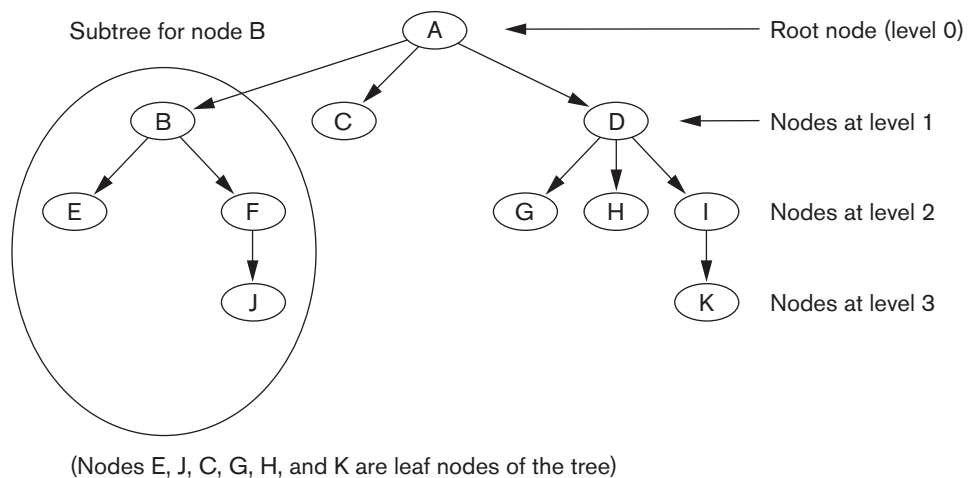
## 17.3 Dynamic Multilevel Indexes Using B-Trees and B<sup>+</sup>-Trees

B-trees and B<sup>+</sup>-trees are special cases of the well-known search data structure known as a **tree**. We briefly introduce the terminology used in discussing tree data structures. A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node. The **level** of a node is always one more than the level of its parent, with the level of the root node being *zero*.<sup>7</sup> A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node *n* and the subtrees of all the child nodes of *n*. Figure 17.7 illustrates a tree data structure. In this figure the root node is A, and its child nodes are B, C, and D. Nodes E, J, C, G, H, and K are leaf nodes. Since the leaf nodes are at different levels of the tree, this tree is called **unbalanced**.

In Section 17.3.1, we introduce search trees and then discuss B-trees, which can be used as dynamic multilevel indexes to guide the search for records in a data file. B-tree nodes are kept between 50 and 100 percent full, and pointers to the data blocks are stored in both internal nodes and leaf nodes of the B-tree structure. In Section 17.3.2 we discuss B<sup>+</sup>-trees, a variation of B-trees in which pointers to the data blocks of a file are stored only in leaf nodes, which can lead to fewer levels and

**Figure 17.7**

A tree data structure that shows an unbalanced tree.



<sup>7</sup>This standard definition of the level of a tree node, which we use throughout Section 17.3, is different from the one we gave for multilevel indexes in Section 17.2.

higher-capacity indexes. In the DBMSs prevalent in the market today, the common structure used for indexing is B<sup>+</sup>-trees.

### 17.3.1 Search Trees and B-Trees

A **search tree** is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields. The multilevel indexes discussed in Section 17.2 can be thought of as a variation of a search tree; each node in the multilevel index can have as many as  $fo$  pointers and  $fo$  key values, where  $fo$  is the index fan-out. The index field values in each node guide us to the next node, until we reach the data file block that contains the required records. By following a pointer, we restrict our search at each level to a subtree of the search tree and ignore all nodes not in this subtree.

**Search Trees.** A search tree is slightly different from a multilevel index. A **search tree of order  $p$**  is a tree such that each node contains *at most*  $p - 1$  search values and  $p$  pointers in the order  $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$ , where  $q \leq p$ . Each  $P_i$  is a pointer to a child node (or a NULL pointer), and each  $K_i$  is a search value from some ordered set of values. All search values are assumed to be unique.<sup>8</sup> Figure 17.8 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

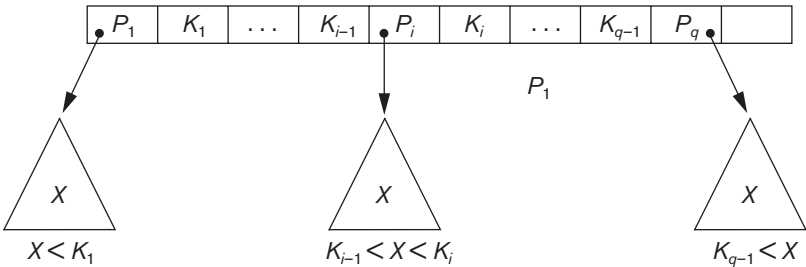
1. Within each node,  $K_1 < K_2 < \dots < K_{q-1}$ .
2. For all values  $X$  in the subtree pointed at by  $P_i$ , we have  $K_{i-1} < X < K_i$  for  $1 < i < q$ ;  $X < K_1$  for  $i = 1$ ; and  $K_{q-1} < X$  for  $i = q$  (see Figure 17.8).

Whenever we search for a value  $X$ , we follow the appropriate pointer  $P_i$  according to the formulas in condition 2 above. Figure 17.9 illustrates a search tree of order  $p = 3$  and integer search values. Notice that some of the pointers  $P_i$  in a node may be NULL pointers.

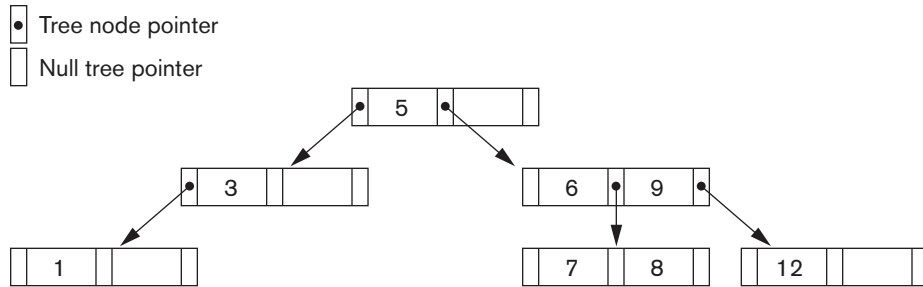
We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the

**Figure 17.8**

A node in a search tree with pointers to subtrees below it.



<sup>8</sup>This restriction can be relaxed. If the index is on a nonkey field, duplicate search values may exist and the node structure and the navigation rules for the tree may be modified.



**Figure 17.9**  
A search tree of order  $p = 3$ .

**search field** (which is the same as the index field if a multilevel index guides the search). Each key value in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted in the file, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is **balanced**, meaning that all of its leaf nodes are at the same level.<sup>9</sup> The tree in Figure 17.7 is not balanced because it has leaf nodes at levels 1, 2, and 3. The goals for balancing a search tree are as follows:

- To guarantee that nodes are evenly distributed, so that the depth of the tree is minimized for the given set of keys and that the tree does not get skewed with some nodes being at very deep levels
- To make the search speed uniform, so that the average time to find any random key is roughly the same

Minimizing the number of levels in the tree is one goal, another implicit goal is to make sure that the index tree does not need too much restructuring as records are inserted into and deleted from the main file. Thus we want the nodes to be as full as possible and do not want any nodes to be empty if there are too many deletions. Record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

**B-Trees.** The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from

<sup>9</sup>The definition of *balanced* is different for binary trees. Balanced binary trees are known as *AVL trees*.

a node that makes it less than half full. More formally, a **B-tree of order  $p$** , when used as an access structure on a *key field* to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree (Figure 17.10(a)) is of the form

$$\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$$

where  $q \leq p$ . Each  $P_i$  is a **tree pointer**—a pointer to another node in the B-tree. Each  $Pr_i$  is a **data pointer**<sup>10</sup>—a pointer to the record whose search key field value is equal to  $K_i$  (or to the data file block containing that record).

2. Within each node,  $K_1 < K_2 < \dots < K_{q-1}$ .
3. For all search key field values  $X$  in the subtree pointed at by  $P_i$  (the  $i$ th subtree, see Figure 17.10(a)), we have:

$$K_{i-1} < X < K_i \text{ for } 1 < i < q; X < K_i \text{ for } i = 1; \text{ and } K_{i-1} < X \text{ for } i = q$$

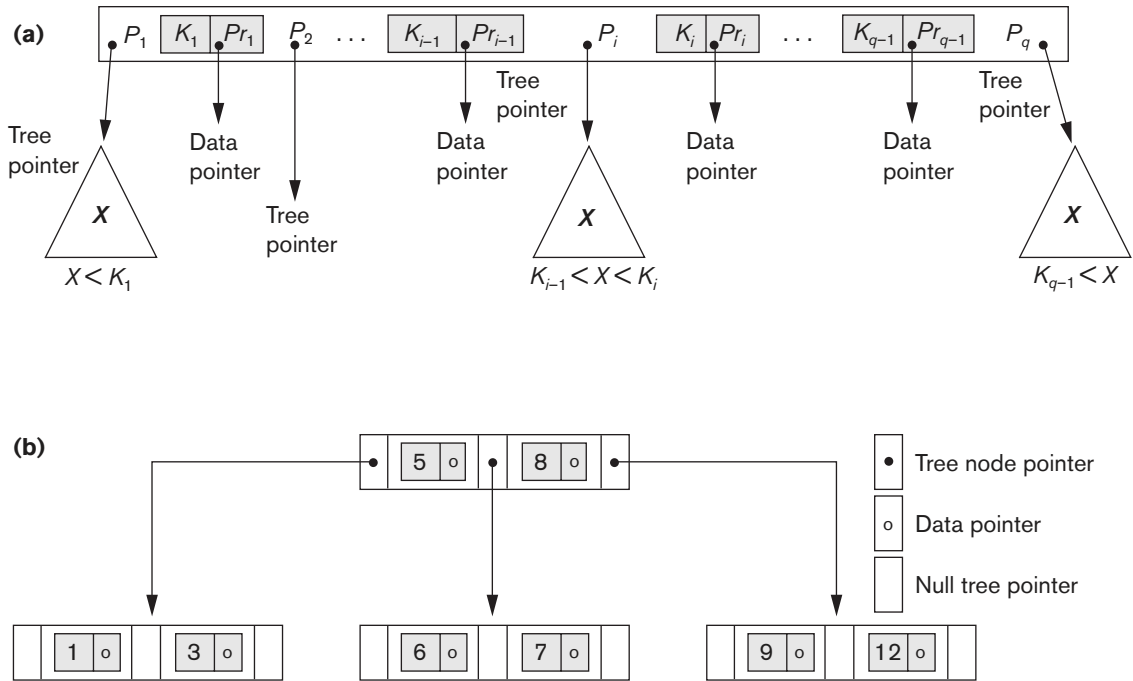
4. Each node has at most  $p$  tree pointers.
5. Each node, except the root and leaf nodes, has at least  $\lceil (p/2) \rceil$  tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
6. A node with  $q$  tree pointers,  $q \leq p$ , has  $q - 1$  search key field values (and hence has  $q - 1$  data pointers).
7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers*  $P_i$  are NULL.

Figure 17.10(b) illustrates a B-tree of order  $p = 3$ . Notice that all search values  $K$  in the B-tree are unique because we assumed that the tree is used as an access structure on a key field. If we use a B-tree *on a nonkey field*, we must change the definition of the file pointers  $Pr_i$  to point to a block—or a cluster of blocks—that contain the pointers to the file records. This extra level of indirection is similar to option 3, discussed in Section 17.1.3, for secondary indexes.

A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with  $p - 1$  search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split. We do not discuss algorithms for B-trees in detail in this text,<sup>11</sup> but we outline search and insertion procedures for B<sup>+</sup>-trees in the next section.

<sup>10</sup>A data pointer is either a block address or a record address; the latter is essentially a block address and a record offset within the block.

<sup>11</sup>For details on insertion and deletion algorithms for B-trees, consult Ramakrishnan and Gehrke (2003).

**Figure 17.10**

B-tree structures. (a) A node in a B-tree with  $q - 1$  search values. (b) A B-tree of order  $p = 3$ . The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. Hence, deletion can reduce the number of tree levels. It has been shown by analysis and simulation that, after numerous random insertions and deletions on a B-tree, the nodes are approximately 69% full when the number of values in the tree stabilizes. This is also true of B<sup>+</sup>-trees. If this happens, node splitting and combining will occur only rarely, so insertion and deletion become quite efficient. If the number of values grows, the tree will expand without a problem—although splitting of nodes may occur, so some insertions will take more time. Each B-tree node can have *at most*  $p$  tree pointers,  $p - 1$  data pointers, and  $p - 1$  search key field values (see Figure 17.10(a)).

In general, a B-tree node may contain additional information needed by the algorithms that manipulate the tree, such as the number of entries  $q$  in the node and a pointer to the parent node. Next, we illustrate how to calculate the number of blocks and levels for a B-tree.

**Example 5.** Suppose that the search field is a nonordering key field, and we construct a B-tree on this field with  $p = 23$ . Assume that each node of the B-tree is 69% full. Each node, on the average, will have  $p * 0.69 = 23 * 0.69$  or approximately



16 pointers and, hence, 15 search key field values. The **average fan-out**  $fo = 16$ . We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

Root:	1 node	15 key entries	16 pointers
Level 1:	16 nodes	240 key entries	256 pointers
Level 2:	256 nodes	3,840 key entries	4,096 pointers
Level 3:	4,096 nodes	61,440 key entries	

At each level, we calculated the number of key entries by multiplying the total number of pointers at the previous level by 15, the average number of entries in each node. Hence, for the given block size (512 bytes), record/data pointer size (7 bytes), tree/block pointer size (6 bytes), and search key field size (9bytes), a two-level B-tree of order 23 with 69% occupancy holds  $3,840 + 240 + 15 = 4,095$  entries on the average; a three-level B-tree holds 65,535 entries on the average.

B-trees are sometimes used as **primary file organizations**. In this case, *whole records* are stored within the B-tree nodes rather than just the <search key, record pointer> entries. This works well for files with a relatively *small number of records* and a *small record size*. Otherwise, the fan-out and the number of levels become too great to permit efficient access.

In summary, B-trees provide a multilevel access structure that is a balanced tree structure in which each node is at least half full. Each node in a B-tree of order  $p$  can have at most  $p - 1$  search values.

### 17.3.2 B<sup>+</sup>-Trees

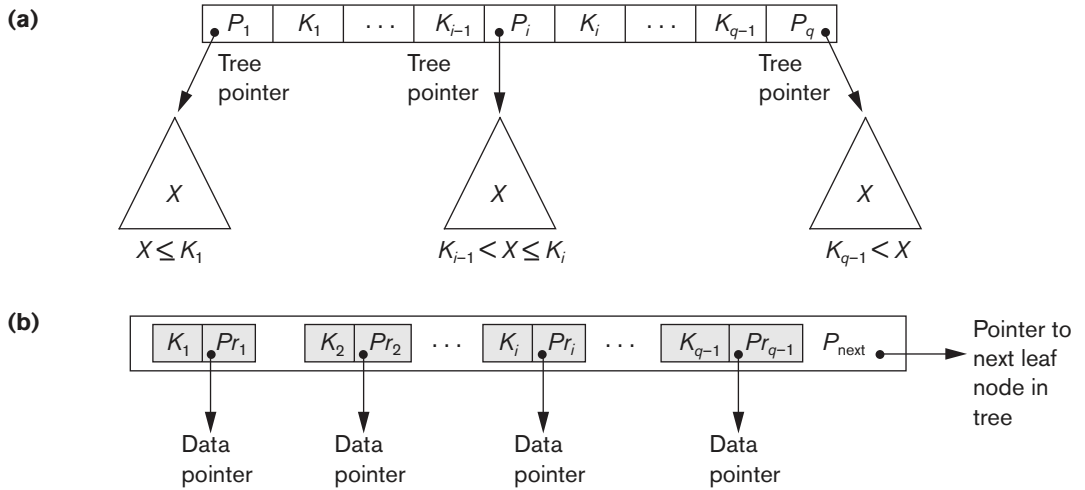
Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B<sup>+</sup>-tree**. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B<sup>+</sup>-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

The leaf nodes of the B<sup>+</sup>-tree are usually linked to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B<sup>+</sup>-tree correspond to the other levels of a multilevel index. Some search field values from the leaf nodes are *repeated* in the internal nodes of the B<sup>+</sup>-tree to guide the search. The structure of the *internal nodes* of a B<sup>+</sup>-tree of order  $p$  (Figure 17.11(a)) is as follows:

1. Each internal node is of the form

$$\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$$

where  $q \leq p$  and each  $P_i$  is a **tree pointer**.

**Figure 17.11**

The nodes of a B<sup>+</sup>-tree. (a) Internal node of a B<sup>+</sup>-tree with  $q - 1$  search values. (b) Leaf node of a B<sup>+</sup>-tree with  $q - 1$  search values and  $q - 1$  data pointers.

2. Within each internal node,  $K_1 < K_2 < \dots < K_{q-1}$ .
3. For all search field values  $X$  in the subtree pointed at by  $P_i$ , we have  $K_{i-1} < X \leq K_i$  for  $1 < i < q$ ;  $X \leq K_i$  for  $i = 1$ ; and  $K_{i-1} < X$  for  $i = q$  (see Figure 17.11(a)).<sup>12</sup>
4. Each internal node has at most  $p$  tree pointers.
5. Each internal node, except the root, has at least  $\lceil (p/2) \rceil$  tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with  $q$  pointers,  $q \leq p$ , has  $q - 1$  search field values.

The structure of the *leaf nodes* of a B<sup>+</sup>-tree of order  $p$  (Figure 17.11(b)) is as follows:

1. Each leaf node is of the form

$$\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{\text{next}} \rangle$$

where  $q \leq p$ , each  $Pr_i$  is a data pointer, and  $P_{\text{next}}$  points to the next *leaf node* of the B<sup>+</sup>-tree.

2. Within each leaf node,  $K_1 \leq K_2 \leq \dots \leq K_{q-1}$ ,  $q \leq p$ .
3. Each  $Pr_i$  is a **data pointer** that points to the record whose search field value is  $K_i$  or to a file block containing the record (or to a block of record pointers that point to records whose search field value is  $K_i$  if the search field is not a key).
4. Each leaf node has at least  $\lceil (p/2) \rceil$  values.
5. All leaf nodes are at the same level.

<sup>12</sup>Our definition follows Knuth (1998). One can define a B<sup>+</sup>-tree differently by exchanging the  $<$  and  $\leq$  symbols ( $K_{i-1} \leq X < K_i$ ;  $K_{q-1} \leq X$ ), but the principles remain the same.

The pointers in internal nodes are *tree pointers* to blocks that are tree nodes, whereas the pointers in leaf nodes are *data pointers* to the data file records or blocks—except for the  $P_{\text{next}}$  pointer, which is a tree pointer to the next leaf node. By starting at the leftmost leaf node, it is possible to traverse leaf nodes as a linked list, using the  $P_{\text{next}}$  pointers. This provides ordered access to the data records on the indexing field. A  $P_{\text{previous}}$  pointer can also be included. For a  $B^+$ -tree on a nonkey field, an extra level of indirection is needed similar to the one shown in Figure 17.5, so the  $Pr$  pointers are block pointers to blocks that contain a set of record pointers to the actual records in the data file, as discussed in option 3 of Section 17.1.3.

Because entries in the *internal nodes* of a  $B^+$ -tree include search values and tree pointers without any data pointers, more entries can be packed into an internal node of a  $B^+$ -tree than for a similar B-tree. Thus, for the same block (node) size, the order  $p$  will be larger for the  $B^+$ -tree than for the B-tree, as we illustrate in Example 6. This can lead to fewer  $B^+$ -tree levels, improving search time. Because the structures for internal and for leaf nodes of a  $B^+$ -tree are different, the order  $p$  can be different. We will use  $p$  to denote the order for *internal nodes* and  $p_{\text{leaf}}$  to denote the order for *leaf nodes*, which we define as being the maximum number of data pointers in a leaf node.

**Example 6.** To calculate the order  $p$  of a  $B^+$ -tree, suppose that the search key field is  $V = 9$  bytes long, the block size is  $B = 512$  bytes, a record pointer is  $Pr = 7$  bytes, and a block pointer/tree pointer is  $P = 6$  bytes. An internal node of the  $B^+$ -tree can have up to  $p$  tree pointers and  $p - 1$  search field values; these must fit into a single block. Hence, we have:

$$\begin{aligned}(p * P) + ((p - 1) * V) &\leq B \\(p * 6) + ((p - 1) * 9) &\leq 512 \\(15 * p) &\leq 512\end{aligned}$$

We can choose  $p$  to be the largest value satisfying the above inequality, which gives  $p = 34$ . This is larger than the value of 23 for the B-tree (it is left to the reader to compute the order of the B-tree assuming same size pointers), resulting in a larger fan-out and more entries in each internal node of a  $B^+$ -tree than in the corresponding B-tree. The leaf nodes of the  $B^+$ -tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order  $p_{\text{leaf}}$  for the leaf nodes can be calculated as follows:

$$\begin{aligned}(p_{\text{leaf}} * (Pr + V)) + P &\leq B \\(p_{\text{leaf}} * (7 + 9)) + 6 &\leq 512 \\(16 * p_{\text{leaf}}) &\leq 506\end{aligned}$$

It follows that each leaf node can hold up to  $p_{\text{leaf}} = 31$  key value/data pointer combinations, assuming that the data pointers are record pointers.

As with the B-tree, we may need additional information—to implement the insertion and deletion algorithms—in each node. This information can include the type of node (internal or leaf), the number of current entries  $q$  in the node, and pointers to the parent and sibling nodes. Hence, before we do the above calculations for  $p$

and  $p_{\text{leaf}}$ , we should reduce the block size by the amount of space needed for all such information. The next example illustrates how we can calculate the number of entries in a B<sup>+</sup>-tree.

**Example 7.** Suppose that we construct a B<sup>+</sup>-tree on the field in Example 6. To calculate the approximate number of entries in the B<sup>+</sup>-tree, we assume that each node is 69% full. On the average, each internal node will have  $34 * 0.69$  or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold  $0.69 * p_{\text{leaf}} = 0.69 * 31$  or approximately 21 data record pointers. A B<sup>+</sup>-tree will have the following average number of entries at each level:

Root:	1 node	22 key entries	23 pointers
Level 1:	23 nodes	506 key entries	529 pointers
Level 2:	529 nodes	11,638 key entries	12,167 pointers
Leaf level:	12,167 nodes	255,507 data record pointers	

For the block size, pointer size, and search field size as in Example 6, a three-level B<sup>+</sup>-tree holds up to 255,507 record pointers, with the average 69% occupancy of nodes. Note that we considered the leaf node differently from the nonleaf nodes and computed the data pointers in the leaf node to be  $12,167 * 21$  based on 69% occupancy of the leaf node, which can hold 31 keys with data pointers. Compare this to the 65,535 entries for the corresponding B-tree in Example 5. Because a B-tree includes a data/record pointer along with each search key at all levels of the tree, it tends to accommodate less number of keys for a given number of index levels. This is the main reason that B<sup>+</sup>-trees are preferred to B-trees as indexes to database files. Most DBMSs, such as Oracle, are creating all indexes as B<sup>+</sup>-trees.

**Search, Insertion, and Deletion with B<sup>+</sup>-Trees.** Algorithm 17.2 outlines the procedure using the B<sup>+</sup>-tree as the access structure to search for a record. Algorithm 17.3 illustrates the procedure for inserting a record in a file with a B<sup>+</sup>-tree access structure. These algorithms assume the existence of a key search field, and they must be modified appropriately for the case of a B<sup>+</sup>-tree on a nonkey field. We illustrate insertion and deletion with an example.

**Algorithm 17.2.** Searching for a Record with Search Key Field Value  $K$ , Using a B<sup>+</sup>-Tree

```

 $n \leftarrow$  block containing root node of B+-tree;
read block  $n$ ;
while ( $n$  is not a leaf node of the B+-tree) do
    begin
         $q \leftarrow$  number of tree pointers in node  $n$ ;
        if  $K \leq n.K_1$  (* $n.K_i$  refers to the  $i$ th search field value in node  $n$ *)
            then  $n \leftarrow n.P_1$  (* $n.P_i$  refers to the  $i$ th tree pointer in node  $n$ *)
        else if  $K > n.K_{q-1}$ 
            then  $n \leftarrow n.P_q$ 
    
```

```

        else begin
            search node  $n$  for an entry  $i$  such that  $n.K_{i-1} < K \leq n.K_i$ ;
             $n \leftarrow n.P_i$ 
        end;

    read block  $n$ 
end;

search block  $n$  for entry  $(K_i, Pr_i)$  with  $K = K_i$  (* search leaf node *)
if found
    then read data file block with address  $Pr_i$  and retrieve record
    else the record with search field value  $K$  is not in the data file;

```

**Algorithm 17.3.** Inserting a Record with Search Key Field Value  $K$  in a  $B^+$ -Tree of Order  $p$

```

 $n \leftarrow$  block containing root node of  $B^+$ -tree;
read block  $n$ ; set stack  $S$  to empty;
while ( $n$  is not a leaf node of the  $B^+$ -tree) do
    begin
        push address of  $n$  on stack  $S$ ;
        (*stack  $S$  holds parent nodes that are needed in case of split*)
         $q \leftarrow$  number of tree pointers in node  $n$ ;
        if  $K \leq n.K_1$  (* $n.K_i$  refers to the  $i$ th search field value in node  $n$ *)
            then  $n \leftarrow n.P_1$  (* $n.P_i$  refers to the  $i$ th tree pointer in node  $n$ *)
            else if  $K \leq n.K_{q-1}$ 
                then  $n \leftarrow n.P_q$ 
            else begin
                search node  $n$  for an entry  $i$  such that  $n.K_{i-1} < K \leq n.K_i$ ;
                 $n \leftarrow n.P_i$ 
            end;

        read block  $n$ 
    end;

search block  $n$  for entry  $(K_i, Pr_i)$  with  $K = K_i$  (*search leaf node  $n$ *)
if found
    then record already in file; cannot insert
    else (*insert entry in  $B^+$ -tree to point to record*)
        begin
            create entry  $(K, Pr)$  where  $Pr$  points to the new record;
            if leaf node  $n$  is not full
                then insert entry  $(K, Pr)$  in correct position in leaf node  $n$ 
            else begin (*leaf node  $n$  is full with  $p_{\text{leaf}}$  record pointers; is split*)
                copy  $n$  to  $temp$  (* $temp$  is an oversize leaf node to hold extra entries*);
                insert entry  $(K, Pr)$  in  $temp$  in correct position;
                (* $temp$  now holds  $p_{\text{leaf}} + 1$  entries of the form  $(K_i, Pr_i)$ *)
                 $new \leftarrow$  a new empty leaf node for the tree;  $new.P_{\text{next}} \leftarrow n.P_{\text{next}}$ ;
                 $j \leftarrow \lceil (p_{\text{leaf}} + 1)/2 \rceil$ ;
                 $n \leftarrow$  first  $j$  entries in  $temp$  (up to entry  $(K_j, Pr_j)$ );  $n.P_{\text{next}} \leftarrow new$ ;
            end
        end

```

```

new ← remaining entries in temp; K ← Kj;
(*now we must move (K, new) and insert in parent internal node;
  however, if parent is full, split may propagate*)
finished ← false;
repeat
if stack S is empty
  then (←no parent node; new root node is created for the tree*)
    begin
      root ← a new empty internal node for the tree;
      root ← <n, K, new>; finished ← true;
    end
  else begin
      n ← pop stack S;
      if internal node n is not full
        then
          begin (*parent node not full; no split*)
            insert (K, new) in correct position in internal node n;
            finished ← true;
          end
        else begin (*internal node n is full with p tree pointers;
          overflow condition; node is split*)
            copy n to temp (*temp is an oversize internal node*);
            insert (K, new) in temp in correct position;
            (*temp now has p + 1 tree pointers*)
            new ← a new empty internal node for the tree;
            j ← ⌊((p + 1)/2)⌋;
            n ← entries up to tree pointer Pj in temp;
            (*n contains <P1, K1, P2, K2, ..., Pj-1, Kj-1, Pj>*)
            new ← entries from tree pointer Pj+1 in temp;
            (*new contains <Pj+1, Kj+1, ..., Kp-1, Pp, Kp, Pp+1>*)
            K ← Kj
            (*now we must move (K, new) and insert in
              parent internal node*)
          end
        end
      until finished
    end;
  end;

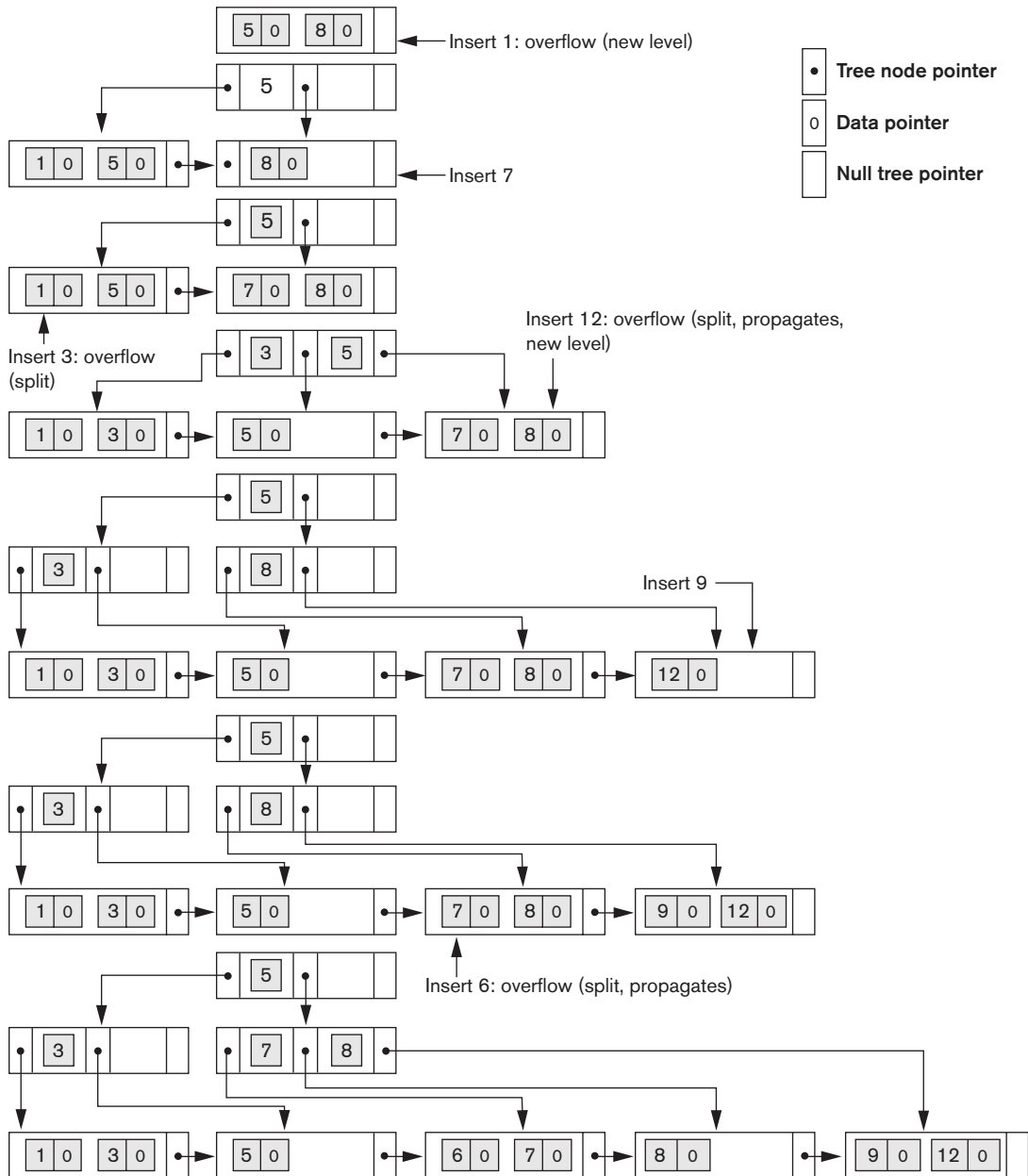
```

Figure 17.12 illustrates insertion of records in a B<sup>+</sup>-tree of order  $p = 3$  and  $p_{\text{leaf}} = 2$ . First, we observe that the root is the only node in the tree, so it is also a leaf node. As soon as more than one level is created, the tree is divided into internal nodes and leaf nodes. Notice that *every key value must exist at the leaf level*, because all data pointers are at the leaf level. However, only some values exist in internal nodes to guide the search. Notice also that every value appearing in an internal node also appears as *the rightmost value* in the leaf level of the subtree pointed at by the tree pointer to the left of the value.

**Figure 17.12**

An example of insertion in a B<sup>+</sup>-tree with  $p = 3$  and  $p_{\text{leaf}} = 2$ .

Insertion sequence: 8, 5, 1, 7, 3, 12, 9, 6



When a *leaf node* is full and a new entry is inserted there, the node *overflows* and must be split. The first  $j = \lceil ((p_{\text{leaf}} + 1)/2) \rceil$  entries in the original node are kept there, and the remaining entries are moved to a new leaf node. The  $j$ th search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent. These must be inserted in the parent node in their correct sequence. If the parent internal node is full, the new value will cause it to overflow also, so it must be split. The entries in the internal node up to  $P_j$ —the  $j$ th tree pointer after inserting the new value and pointer, where  $j = \lfloor ((p + 1)/2) \rfloor$ —are kept, whereas the  $j$ th search value is moved to the parent, not replicated. A new internal node will hold the entries from  $P_{j+1}$  to the end of the entries in the node (see Algorithm 17.3). This splitting can propagate all the way up to create a new root node and hence a new level for the B<sup>+</sup>-tree.

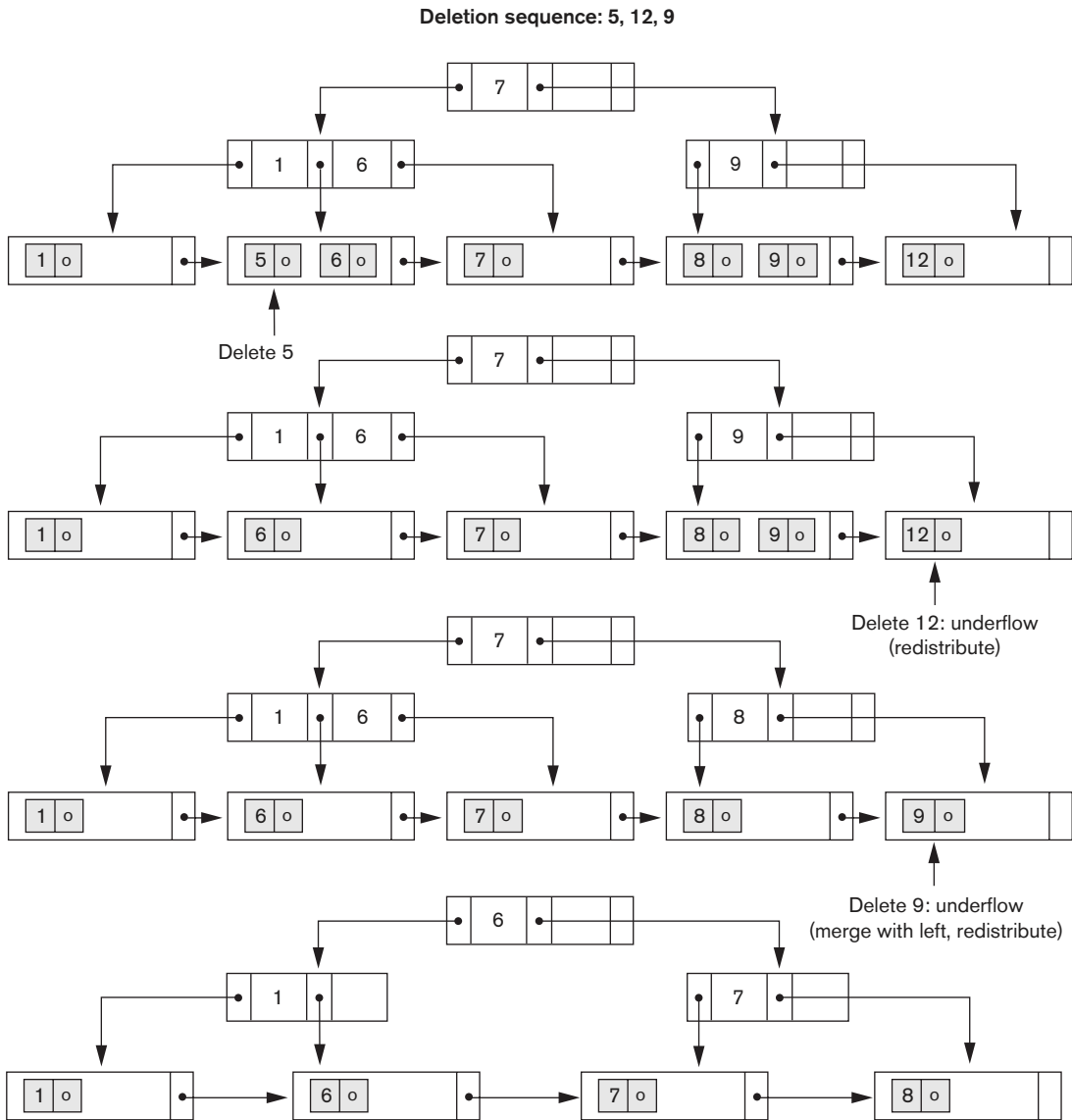
Figure 17.13 illustrates deletion from a B<sup>+</sup>-tree. When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree. Deletion may cause **underflow** by reducing the number of entries in the leaf node to below the minimum required. In this case, we try to find a sibling leaf node—a leaf node directly to the left or to the right of the node with underflow—and redistribute the entries among the node and its **sibling** so that both are at least half full; otherwise, the node is merged with its siblings and the number of leaf nodes is reduced. A common method is to try to **redistribute** entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is made. If this is also not possible, the three nodes are merged into two leaf nodes. In such a case, underflow may propagate to **internal** nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.

Notice that implementing the insertion and deletion algorithms may require parent and sibling pointers for each node, or the use of a stack as in Algorithm 17.3. Each node should also include the number of entries in it and its type (leaf or internal). Another alternative is to implement insertion and deletion as recursive procedures.<sup>13</sup>

**Variations of B-Trees and B<sup>+</sup>-Trees.** To conclude this section, we briefly mention some variations of B-trees and B<sup>+</sup>-trees. In some cases, constraint 5 on the B-tree (or for the internal nodes of the B<sup>+</sup>-tree, except the root node), which requires each node to be at least half full, can be changed to require each node to be at least two-thirds full. In this case the B-tree has been called a **B\*-tree**. In general, some systems allow the user to choose a **fill factor** between 0.5 and 1.0, where the latter means that the B-tree (index) nodes are to be completely full. It is also possible to specify two fill factors for a B<sup>+</sup>-tree: one for the leaf level and one for the internal nodes of the tree. When the index is first constructed, each node is filled up

<sup>13</sup>For more details on insertion and deletion algorithms for B<sup>+</sup>-trees, consult Ramakrishnan and Gehrke (2003).





**Figure 17.13**  
An example of deletion from a B<sup>+</sup>-tree.

to approximately the fill factors specified. Some investigators have suggested relaxing the requirement that a node be half full, and instead allow a node to become completely empty before merging, to simplify the deletion algorithm. Simulation studies show that this does not waste too much additional space under randomly distributed insertions and deletions.

## 17.4 Indexes on Multiple Keys

In our discussion so far, we have assumed that the primary or secondary keys on which files were accessed were single attributes (fields). In many retrieval and update requests, multiple attributes are involved. If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes.

For example, consider an EMPLOYEE file containing attributes Dno (department number), Age, Street, City, Zip\_code, Salary and Skill\_code, with the key of Ssn (Social Security number). Consider the query: *List the employees in department number 4 whose age is 59.* Note that both Dno and Age are nonkey attributes, which means that a search value for either of these will point to multiple records. The following alternative search strategies may be considered:

1. Assuming Dno has an index, but Age does not, access the records having Dno = 4 using the index, and then select from among them those records that satisfy Age = 59.
2. Alternately, if Age is indexed but Dno is not, access the records having Age = 59 using the index, and then select from among them those records that satisfy Dno = 4.
3. If indexes have been created on both Dno and Age, both indexes may be used; each gives a set of records or a set of pointers (to blocks or records). An intersection of these sets of records or pointers yields those records or pointers that satisfy both conditions.

All of these alternatives eventually give the correct result. However, if the set of records that meet each condition (Dno = 4 or Age = 59) individually are large, yet only a few records satisfy the combined condition, then none of the above is an efficient technique for the given search request. Note also that queries such as “find the minimum or maximum age among all employees” can be answered just by using the index on Age, without going to the data file. Finding the maximum or minimum age within Dno = 4, however, would not be answerable just by processing the index alone. Also, listing the departments in which employees with Age = 59 work will also not be possible by processing just the indexes. A number of possibilities exist that would treat the combination <Dno, Age> or <Age, Dno> as a search key made up of multiple attributes. We briefly outline these techniques in the following sections. We will refer to keys containing multiple attributes as **composite keys**.

### 17.4.1 Ordered Index on Multiple Attributes

All the discussion in this chapter so far still applies if we create an index on a search key field that is a combination of <Dno, Age>. The search key is a pair of values <4, 59> in the above example. In general, if an index is created on attributes <A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>>, the search key values are tuples with *n* values: <v<sub>1</sub>, v<sub>2</sub>, ..., v<sub>n</sub>>.

A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of the department keys for department number

3 precede those for department number 4. Thus  $\langle 3, n \rangle$  precedes  $\langle 4, m \rangle$  for any values of  $m$  and  $n$ . The ascending key order for keys with  $\text{Dno} = 4$  would be  $\langle 4, 18 \rangle$ ,  $\langle 4, 19 \rangle$ ,  $\langle 4, 20 \rangle$ , and so on. Lexicographic ordering works similarly to ordering of character strings. An index on a composite key of  $n$  attributes works similarly to any index discussed in this chapter so far.

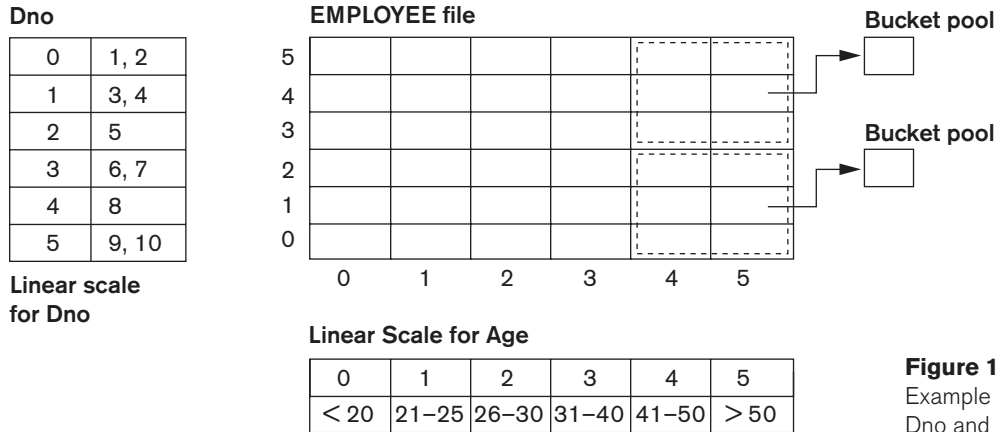
### 17.4.2 Partitioned Hashing

Partitioned hashing is an extension of static external hashing (Section 16.8.2) that allows access on multiple keys. It is suitable only for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of  $n$  components, the hash function is designed to produce a result with  $n$  separate hash addresses. The bucket address is a concatenation of these  $n$  addresses. It is then possible to search for the required composite search key by looking up the appropriate buckets that match the parts of the address in which we are interested.

For example, consider the composite search key  $\langle \text{Dno}, \text{Age} \rangle$ . If  $\text{Dno}$  and  $\text{Age}$  are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that  $\text{Dno} = 4$  has a hash address '100' and  $\text{Age} = 59$  has hash address '10101'. Then to search for the combined search value,  $\text{Dno} = 4$  and  $\text{Age} = 59$ , one goes to bucket address 100 10101; just to search for all employees with  $\text{Age} = 59$ , all buckets (eight of them) will be searched whose addresses are '000 10101', '001 10101', ... and so on. An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high-order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes. Additionally, most hash functions do not maintain records in order by the key being hashed. Hence, accessing records in lexicographic order by a combination of attributes such as  $\langle \text{Dno}, \text{Age} \rangle$  used as a key would not be straightforward or efficient.

### 17.4.3 Grid Files

Another alternative is to organize the EMPLOYEE file as a grid file. If we want to access a file on two keys, say  $\text{Dno}$  and  $\text{Age}$  as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes. Figure 17.14 shows a grid array for the EMPLOYEE file with one linear scale for  $\text{Dno}$  and another for the  $\text{Age}$  attribute. The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for  $\text{Dno}$  has  $\text{Dno} = 1, 2$  combined as one value 0 on the scale, whereas  $\text{Dno} = 5$  corresponds to the value 2 on that scale. Similarly,  $\text{Age}$  is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age. The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored. Figure 17.14 also shows the assignment of cells to buckets (only partially).



**Figure 17.14**  
Example of a grid array on Dno and Age attributes.

Thus our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket. This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. If a range query corresponds to a match on some of the grid cells, it can be processed by accessing exactly the buckets for those grid cells. For example, a query for Dno ≤ 5 and Age > 40 refers to the data in the top bucket shown in Figure 17.14.

The grid file concept can be applied to any number of search keys. For example, for  $n$  search keys, the grid array would have  $n$  dimensions. The grid array thus allows a partitioning of the file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions. Grid files perform well in terms of reduction in time for multiple key access. However, they represent a space overhead in terms of the grid array structure. Moreover, with dynamic files, a frequent reorganization of the file adds to the maintenance cost.<sup>14</sup>

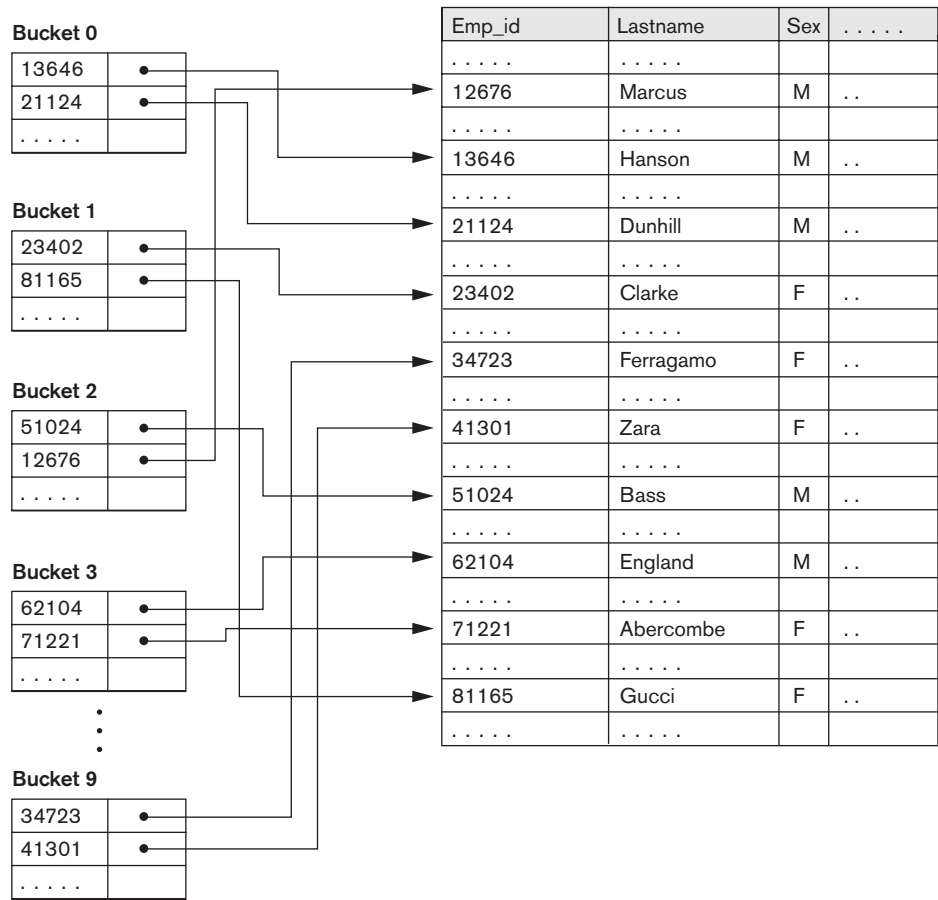
## 17.5 Other Types of Indexes

### 17.5.1 Hash Indexes

It is also possible to create access structures similar to indexes that are based on *hashing*. The **hash index** is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization. The index entries are of the type  $\langle K, Pr \rangle$  or  $\langle K, P \rangle$ , where  $Pr$  is a pointer to the record containing the key, or  $P$  is a pointer to the block containing the record for that key. The index file with these index entries can be organized as a dynamically expandable hash file, using one of the techniques described in Section 16.8.3; searching for an entry uses the hash search algorithm on  $K$ . Once an entry is found, the pointer  $Pr$

<sup>14</sup>Insertion/deletion algorithms for grid files may be found in Nievergelt et al. (1984).

**Figure 17.15**  
Hash-based  
indexing.



(or *P*) is used to locate the corresponding record in the data file. Figure 17.15 illustrates a hash index on the *Emp\_id* field for a file that has been stored as a sequential file ordered by Name. The *Emp\_id* is hashed to a bucket number by using a hashing function: the sum of the digits of *Emp\_id* modulo 10. For example, to find *Emp\_id* 51024, the hash function results in bucket number 2; that bucket is accessed first. It contains the index entry < 51024, *Pr* >; the pointer *Pr* leads us to the actual record in the file. In a practical application, there may be thousands of buckets; the bucket number, which may be several bits long, would be subjected to the directory schemes discussed in the context of dynamic hashing in Section 16.8.3. Other search structures can also be used as indexes.

### 17.5.2 Bitmap Indexes

The **bitmap index** is another popular data structure that facilitates querying on multiple keys. Bitmap indexing is used for relations that contain a large number of rows. It creates an index for one or more columns, and each value or value range in

**EMPLOYEE**

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

**Bitmap index for Sex**

M	F
10100110	01011001

**Bitmap index for Zipcode**

Zipcode 19046	Zipcode 30022	Zipcode 94040
00101100	01010010	10000001

**Figure 17.16**

Bitmap indexes for Sex and Zipcode.

those columns is indexed. Typically, a bitmap index is created for those columns that contain a fairly small number of unique values. To build a bitmap index on a set of records in a relation, the records must be numbered from 0 to  $n$  with an id (a record id or a row id) that can be mapped to a physical address made of a block number and a record offset within the block.

A bitmap index is built on **one particular value** of a particular field (the column in a relation) and is just an array of bits. Thus, for a given field, there is one separate bitmap index (or a vector) maintained corresponding to each unique value in the database. Consider a bitmap index for the column  $C$  and a value  $V$  for that column. For a relation with  $n$  rows, it contains  $n$  bits. The  $i$ th bit is set to 1 if the row  $i$  has the value  $V$  for column  $C$ ; otherwise it is set to a 0. If  $C$  contains the valueset  $\langle v_1, v_2, \dots, v_m \rangle$  with  $m$  distinct values, then  $m$  bitmap indexes would be created for that column. Figure 17.16 shows the relation EMPLOYEE with columns Emp\_id, Lname, Sex, Zipcode, and Salary\_grade (with just eight rows for illustration) and a bitmap index for the Sex and Zipcode columns. As an example, if the bitmap for Sex = F, the bits for Row\_ids 1, 3, 4, and 7 are set to 1, and the rest of the bits are set to 0, the bitmap indexes could have the following query applications:

- For the query  $C_1 = V_1$ , the corresponding bitmap for value  $V_1$  returns the Row\_ids containing the rows that qualify.
- For the query  $C_1 = V_1$  and  $C_2 = V_2$  (a multikey search request), the two corresponding bitmaps are retrieved and intersected (logically AND-ed) to yield the set of Row\_ids that qualify. In general,  $k$  bitvectors can be intersected to deal with  $k$  equality conditions. Complex AND-OR conditions can also be supported using bitmap indexing.
- For the query  $C_1 = V_1$  or  $C_2 = V_2$  or  $C_3 = V_3$  (a multikey search request), the three corresponding bitmaps for three different attributes are retrieved and unioned (logically OR-ed) to yield the set of Row ids that qualify.

- To retrieve a count of rows that qualify for the condition  $C_1 = V_1$ , the “1” entries in the corresponding bitvector are counted.
- Queries with negation, such as  $C_1 \neg = V_1$ , can be handled by applying the Boolean *complement* operation on the corresponding bitmap.

Consider the example relation EMPLOYEE in Figure 17.16 with bitmap indexes on Sex and Zipcode. To find employees with Sex = F and Zipcode = 30022, we intersect the bitmaps “01011001” and “01010010” yielding Row\_ids 1 and 3. Employees who do not live in Zipcode = 94040 are obtained by complementing the bitvector “10000001” and yields Row\_ids 1 through 6. In general, if we assume uniform distribution of values for a given column, and if one column has 5 distinct values and another has 10 distinct values, the join condition on these two can be considered to have a selectivity of  $1/50$  ( $= 1/5 * 1/10$ ). Hence, only about 2% of the records would actually have to be retrieved. If a column has only a few values, like the Sex column in Figure 17.16, retrieval of the Sex = M condition on average would retrieve 50% of the rows; in such cases, it is better to do a complete scan rather than use bitmap indexing.

In general, bitmap indexes are efficient in terms of the storage space that they need. If we consider a file of 1 million rows (records) with record size of 100 bytes per row, each bitmap index would take up only one bit per row and hence would use 1 million bits or 125 Kbytes. Suppose this relation is for 1 million residents of a state, and they are spread over 200 ZIP Codes; the 200 bitmaps over Zipcodes contribute 200 bits (or 25 bytes) worth of space per row; hence, the 200 bitmaps occupy only 25% as much space as the data file. They allow an exact retrieval of all residents who live in a given ZIP Code by yielding their Row\_ids.

When records are deleted, renumbering rows and shifting bits in bitmaps becomes expensive. Another bitmap, called the **existence bitmap**, can be used to avoid this expense. This bitmap has a 0 bit for the rows that have been deleted but are still physically present and a 1 bit for rows that actually exist. Whenever a row is inserted in the relation, an entry must be made in all the bitmaps of all the columns that have a bitmap index; rows typically are appended to the relation or may replace deleted rows to minimize the impact on the reorganization of the bitmaps. This process still constitutes an indexing overhead.

Large bitvectors are handled by treating them as a series of 32-bit or 64-bit vectors, and corresponding AND, OR, and NOT operators are used from the instruction set to deal with 32- or 64-bit input vectors in a single instruction. This makes bitvector operations computationally very efficient.

**Bitmaps for B<sup>+</sup>-Tree Leaf Nodes.** Bitmaps can be used on the leaf nodes of B<sup>+</sup>-tree indexes as well as to point to the set of records that contain each specific value of the indexed field in the leaf node. When the B<sup>+</sup>-tree is built on a nonkey search field, the leaf record must contain a list of record pointers alongside each value of the indexed attribute. For values that occur very frequently, that is, in a large percentage of the relation, a bitmap index may be stored instead of the pointers. As an

example, for a relation with  $n$  rows, suppose a value occurs in 10% of the file records. A bitvector would have  $n$  bits, having the “1” bit for those `Row_ids` that contain that search value, which is  $n/8$  or  $0.125n$  bytes in size. If the record pointer takes up 4 bytes (32 bits), then the  $n/10$  record pointers would take up  $4 * n/10$  or  $0.4n$  bytes. Since  $0.4n$  is more than 3 times larger than  $0.125n$ , it is better to store the bitmap index rather than the record pointers. Hence for search values that occur more frequently than a certain ratio (in this case that would be  $1/32$ ), it is beneficial to use bitmaps as a compressed storage mechanism for representing the record pointers in  $B^+$ -trees that index a nonkey field.

### 17.5.3 Function-Based Indexing

In this section, we discuss a new type of indexing, called **function-based indexing**, that has been introduced in the Oracle relational DBMS as well as in some other commercial products.<sup>15</sup>

The idea behind function-based indexing is to create an index such that the value that results from applying some function on a field or a collection of fields becomes the key to the index. The following examples show how to create and use function-based indexes.

**Example 1.** The following statement creates a function-based index on the `EMPLOYEE` table based on an uppercase representation of the `Lname` column, which can be entered in many ways but is always queried by its uppercase representation.

```
CREATE INDEX upper_ix ON Employee (UPPER(Lname));
```

This statement will create an index based on the function `UPPER(Lname)`, which returns the last name in uppercase letters; for example, `UPPER('Smith')` will return `'SMITH'`.

Function-based indexes ensure that Oracle Database system will use the index rather than perform a full table scan, even when a function is used in the search predicate of a query. For example, the following query will use the index:

```
SELECT First_name, Lname
FROM Employee
WHERE UPPER(Lname)= "SMITH".
```

Without the function-based index, an Oracle Database might perform a full table scan, since a  $B^+$ -tree index is searched only by using the column value directly; the use of any function on a column prevents such an index from being used.

**Example 2.** In this example, the `EMPLOYEE` table is supposed to contain two fields—`salary` and `commission_pct` (commission percentage)—and an index is being created on the sum of salary and commission based on the `commission_pct`.

```
CREATE INDEX income_ix
ON Employee(Salary + (Salary*Commission_pct));
```

<sup>15</sup>Rafi Ahmed contributed most of this section.



The following query uses the `income_ix` index even though the fields `salary` and `commission_pct` are occurring in the reverse order in the query when compared to the index definition.

```
SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000;
```

**Example 3.** This is a more advanced example of using function-based indexing to define conditional uniqueness. The following statement creates a unique function-based index on the `ORDERS` table that prevents a customer from taking advantage of a promotion id (“blowout sale”) more than once. It creates a composite index on the `Customer_id` and `Promotion_id` fields together, and it allows only one entry in the index for a given `Customer_id` with the `Promotion_id` of “2” by declaring it as a unique index.

```
CREATE UNIQUE INDEX promo_ix ON Orders
(CASE WHEN Promotion_id = 2 THEN Customer_id ELSE NULL END,
CASE WHEN Promotion_id = 2 THEN Promotion_id ELSE NULL END);
```

Note that by using the `CASE` statement, the objective is to remove from the index any rows where `Promotion_id` is not equal to 2. Oracle Database does not store in the  $B^+$ -tree index any rows where all the keys are `NULL`. Therefore, in this example, we map both `Customer_id` and `Promotion_id` to `NULL` unless `Promotion_id` is equal to 2. The result is that the index constraint is violated only if `Promotion_id` is equal to 2, for two (attempted insertions of) rows with the same `Customer_id` value.

## 17.6 Some General Issues Concerning Indexing

### 17.6.1 Logical versus Physical Indexes

In the earlier discussion, we have assumed that the index entries  $\langle K, Pr \rangle$  (or  $\langle K, P \rangle$ ) always include a physical pointer  $Pr$  (or  $P$ ) that specifies the physical record address on disk as a block number and offset. This is sometimes called a **physical index**, and it has the disadvantage that the pointer must be changed if the record is moved to another disk location. For example, suppose that a primary file organization is based on linear hashing or extendible hashing; then, each time a bucket is split, some records are allocated to new buckets and hence have new physical addresses. If there was a secondary index on the file, the pointers to those records would have to be found and updated, which is a difficult task.

To remedy this situation, we can use a structure called a **logical index**, whose index entries are of the form  $\langle K, K_p \rangle$ . Each entry has one value  $K$  for the secondary indexing field matched with the value  $K_p$  of the field used for the primary file organization. By searching the secondary index on the value of  $K$ , a program can locate the corresponding value of  $K_p$  and use this to access the record through the primary file organization, using a primary index if available. Logical indexes thus introduce an

additional level of indirection between the access structure and the data. They are used when physical record addresses are expected to change frequently. The cost of this indirection is the extra search based on the primary file organization.

### 17.6.2 Index Creation

Many RDBMSs have a similar type of command for creating an index, although it is not part of the SQL standard. The general form of this command is:

```
CREATE [ UNIQUE ] INDEX <index name>
ON <table name> ( <column name> [ <order> ] { , <column name> [ <order> ] } )
[ CLUSTER ] ;
```

The keywords **UNIQUE** and **CLUSTER** are optional. The keyword **CLUSTER** is used when the index to be created should also sort the data file records on the indexing attribute. Thus, specifying **CLUSTER** on a key (unique) attribute would create some variation of a primary index, whereas specifying **CLUSTER** on a nonkey (nonunique) attribute would create some variation of a clustering index. The value for **<order>** can be either **ASC** (ascending) or **DESC** (descending), and it specifies whether the data file should be ordered in ascending or descending values of the indexing attribute. The default is **ASC**. For example, the following would create a clustering (ascending) index on the nonkey attribute **Dno** of the **EMPLOYEE** file:

```
CREATE INDEX DnoIndex
ON EMPLOYEE (Dno)
CLUSTER ;
```

**Index Creation Process:** In many systems, an index is not an integral part of the data file but can be created and discarded dynamically. That is why it is often called an *access structure*. Whenever we expect to access a file frequently based on some search condition involving a particular field, we can request the DBMS to create an index on that field as shown above for the **DnoIndex**. Usually, a secondary index is created to avoid physical ordering of the records in the data file on disk.

The main advantage of secondary indexes is that—theoretically, at least—they can be created in conjunction with *virtually any primary record organization*. Hence, a secondary index could be used to complement other primary access methods such as ordering or hashing, or it could even be used with mixed files. To create a  $B^+$ -tree secondary index on some field of a file, if the file is large and contains millions of records, neither the file nor the index would fit in main memory. Insertion of a large number of entries into the index is done by a process called **bulk loading** the index. We must go through all records in the file to create the entries at the leaf level of the tree. These entries are then sorted and filled according to the specified fill factor; simultaneously, the other index levels are created. It is more expensive and much harder to create primary indexes and clustering indexes dynamically, because the records of the data file must be physically sorted on disk in order of the indexing field. However, some systems allow users to create these indexes dynamically on their files by sorting the file during index creation.

**Indexing of Strings:** There are a couple of issues that are of particular concern when indexing strings. Strings can be variable length (e.g., VARCHAR data type in SQL; see Chapter 6) and strings may be too long limiting the fan-out. If a B<sup>+</sup>-tree index is to be built with a string as a search key, there may be an uneven number of keys per index node and the fan-out may vary. Some nodes may be forced to split when they become full regardless of the number of keys in them. The technique of **prefix compression** alleviates the situation. Instead of storing the entire string in the intermediate nodes, it stores only the prefix of the search key adequate to distinguish the keys that are being separated and directed to the subtree. For example, if Lastname was a search key and we were looking for “Navathe”, the nonleaf node may contain “Nac” for Nachamkin and “Nay” for Nayuddin as the two keys on either side of the subtree pointer that we need to follow.

### 17.6.3 Tuning Indexes

The initial choice of indexes may have to be revised for the following reasons:

- Certain queries may take too long to run for lack of an index.
- Certain indexes may not get utilized at all.
- Certain indexes may undergo too much updating because the index is on an attribute that undergoes frequent changes.

Most DBMSs have a command or trace facility, which can be used by the DBA to ask the system to show how a query was executed—what operations were performed in what order and what secondary access structures (indexes) were used. By analyzing these execution plans (we will discuss this term further in Chapter 18), it is possible to diagnose the causes of the above problems. Some indexes may be dropped and some new indexes may be created based on the tuning analysis.

The goal of tuning is to dynamically evaluate the requirements, which sometimes fluctuate seasonally or during different times of the month or week, and to reorganize the indexes and file organizations to yield the best overall performance. Dropping and building new indexes is an overhead that can be justified in terms of performance improvements. Updating of a table is generally suspended while an index is dropped or created; this loss of service must be accounted for.

Besides dropping or creating indexes and changing from a nonclustered to a clustered index and vice versa, **rebuilding the index** may improve performance. Most RDBMSs use B<sup>+</sup>-trees for an index. If there are many deletions on the index key, index pages may contain wasted space, which can be claimed during a rebuild operation. Similarly, too many insertions may cause overflows in a clustered index that affect performance. Rebuilding a clustered index amounts to reorganizing the entire table ordered on that key.

The available options for indexing and the way they are defined, created, and reorganized vary from system to system. As an illustration, consider the sparse and dense indexes we discussed in Section 17.1. A sparse index such as a primary index will have one index pointer for each page (disk block) in the data file; a

dense index such as a unique secondary index will have an index pointer for each record. Sybase provides clustering indexes as sparse indexes in the form of B<sup>+</sup>-trees, whereas INGRES provides sparse clustering indexes as ISAM files and dense clustering indexes as B<sup>+</sup>-trees. In some versions of Oracle and DB2, the option of setting up a clustering index is limited to a dense index, and the DBA has to work with this limitation.

#### 17.6.4 Additional Issues Related to Storage of Relations and Indexes

**Using an Index for Managing Constraints and Duplicates:** It is common to use an index to enforce a *key constraint* on an attribute. While searching the index to insert a new record, it is straightforward to check at the same time whether another record in the file—and hence in the index tree—has the same key attribute value as the new record. If so, the insertion can be rejected.

If an index is created on a nonkey field, *duplicates* occur; handling of these duplicates is an issue the DBMS product vendors have to deal with and affects data storage as well as index creation and management. Data records for the duplicate key may be contained in the same block or may span multiple blocks where many duplicates are possible. Some systems add a row id to the record so that records with duplicate keys have their own unique identifiers. In such cases, the B<sup>+</sup>-tree index may regard a <key, Row\_id> combination as the de facto key for the index, turning the index into a unique index with no duplicates. The deletion of a key *K* from such an index would involve deleting all occurrences of that key *K*—hence the deletion algorithm has to account for this.

In actual DBMS products, deletion from B<sup>+</sup>-tree indexes is also handled in various ways to improve performance and response times. Deleted records may be marked as deleted and the corresponding index entries may also not be removed until a garbage collection process reclaims the space in the data file; the index is rebuilt online after garbage collection.

**Inverted Files and Other Access Methods:** A file that has a secondary index on every one of its fields is often called a **fully inverted file**. Because all indexes are secondary, new records are inserted at the end of the file; therefore, the data file itself is an unordered (heap) file. The indexes are usually implemented as B<sup>+</sup>-trees, so they are updated dynamically to reflect insertion or deletion of records. Some commercial DBMSs, such as Software AG's Adabas, use this method extensively.

We referred to the popular IBM file organization called ISAM in Section 17.2. Another IBM method, the **virtual storage access method (VSAM)**, is somewhat similar to the B<sup>+</sup>-tree access structure and is still being used in many commercial systems.

**Using Indexing Hints in Queries:** DBMSs such as Oracle have a provision for allowing hints in queries that are suggested alternatives or indicators to the query

processor and optimizer for expediting query execution. One form of hints is called indexing hints; these hints suggest the use of an index to improve the execution of a query. The hints appear as a special comment (which is preceded by `+`) and they override all optimizer decisions, but they may be ignored by the optimizer if they are invalid, irrelevant, or improperly formulated. We do not get into a detailed discussion of indexing hints, but illustrate with an example query.

For example, to retrieve the SSN, Salary, and department number for employees working in department numbers with Dno less than 10:

```
SELECT /*+ INDEX (EMPLOYEE emp_dno_index) */ Emp_ssn, Salary, Dno
FROM EMPLOYEE
WHERE Dno < 10;
```

The above query includes a hint to use a valid index called `emp_dno_index` (which is an index on the `EMPLOYEE` relation on `Dno`).

**Column-Based Storage of Relations:** There has been a recent trend to consider a column-based storage of relations as an alternative to the traditional way of storing relations row by row. Commercial relational DBMSs have offered B<sup>+</sup>-tree indexing on primary as well as secondary keys as an efficient mechanism to support access to data by various search criteria and the ability to write a row or a set of rows to disk at a time to produce write-optimized systems. For data warehouses (to be discussed in Chapter 29), which are read-only databases, the column-based storage offers particular advantages for read-only queries. Typically, the column-store RDBMSs consider storing each column of data individually and afford performance advantages in the following areas:

- Vertically partitioning the table column by column, so that a two-column table can be constructed for every attribute and thus only the needed columns can be accessed
- Using column-wise indexes (similar to the bitmap indexes discussed in Section 17.5.2) and join indexes on multiple tables to answer queries without having to access the data tables
- Using materialized views (see Chapter 7) to support queries on multiple columns

Column-wise storage of data affords additional freedom in the creation of indexes, such as the bitmap indexes discussed earlier. The same column may be present in multiple projections of a table and indexes may be created on each projection. To store the values in the same column, strategies for data compression, null-value suppression, dictionary encoding techniques (where distinct values in the column are assigned shorter codes), and run-length encoding techniques have been devised. MonetDB/X100, C-Store, and Vertica are examples of such systems. Some popular systems (like Cassandra, Hbase, and Hypertable) have used column-based storage effectively with the concept of **wide column-stores**. The storage of data in such systems will be explained in the context of NOSQL systems that we will discuss in Chapter 24.

## 17.7 Physical Database Design in Relational Databases

In this section, we discuss the physical design factors that affect the performance of applications and transactions, and then we comment on the specific guidelines for RDBMSs in the context of what we discussed in Chapter 16 and this chapter so far.

### 17.7.1 Factors That Influence Physical Database Design

Physical design is an activity where the goal is not only to create the appropriate structuring of data in storage, but also to do so in a way that guarantees good performance. For a given conceptual schema, there are many physical design alternatives in a given DBMS. It is not possible to make meaningful physical design decisions and performance analyses until the database designer knows the mix of queries, transactions, and applications that are expected to run on the database. This is called the **job mix** for the particular set of database system applications. The database administrators/designers must analyze these applications, their expected frequencies of invocation, any timing constraints on their execution speed, the expected frequency of update operations, and any unique constraints on attributes. We discuss each of these factors next.

**A. Analyzing the Database Queries and Transactions.** Before undertaking the physical database design, we must have a good idea of the intended use of the database by defining in a high-level form the queries and transactions that are expected to run on the database. For each **retrieval query**, the following information about the query would be needed:

1. The files (relations) that will be accessed by the query
2. The attributes on which any selection conditions for the query are specified
3. Whether the selection condition is an equality, inequality, or a range condition
4. The attributes on which any join conditions or conditions to link multiple tables or objects for the query are specified
5. The attributes whose values will be retrieved by the query

The attributes listed in items 2 and 4 above are candidates for the definition of access structures, such as indexes, hash keys, or sorting of the file.

For each **update operation** or **update transaction**, the following information would be needed:

1. The files that will be updated
2. The type of operation on each file (insert, update, or delete)
3. The attributes on which selection conditions for a delete or update are specified
4. The attributes whose values will be changed by an update operation

Again, the attributes listed in item 3 are candidates for access structures on the files, because they would be used to locate the records that will be updated or deleted. On

the other hand, the attributes listed in item 4 are candidates for *avoiding an access structure*, since modifying them will require updating the access structures.

**B. Analyzing the Expected Frequency of Invocation of Queries and Transactions.** Besides identifying the characteristics of expected retrieval queries and update transactions, we must consider their expected rates of invocation. This frequency information, along with the attribute information collected on each query and transaction, is used to compile a cumulative list of the expected frequency of use for all queries and transactions. This is expressed as the expected frequency of using each attribute in each file as a selection attribute or a join attribute, over all the queries and transactions. Generally, for large volumes of processing, the informal *80–20 rule* can be used: approximately 80% of the processing is accounted for by only 20% of the queries and transactions. Therefore, in practical situations, it is rarely necessary to collect exhaustive statistics and invocation rates on all the queries and transactions; it is sufficient to determine the 20% or so most important ones.

**C. Analyzing the Time Constraints of Queries and Transactions.** Some queries and transactions may have stringent performance constraints. For example, a transaction may have the constraint that it should terminate within 5 seconds on 95% of the occasions when it is invoked, and that it should never take more than 20 seconds. Such timing constraints place further priorities on the attributes that are candidates for access paths. The selection attributes used by queries and transactions with time constraints become higher-priority candidates for primary access structures for the files, because the primary access structures are generally the most efficient for locating records in a file.

**D. Analyzing the Expected Frequencies of Update Operations.** A minimum number of access paths should be specified for a file that is frequently updated, because updating the access paths themselves slows down the update operations. For example, if a file that has frequent record insertions has 10 indexes on 10 different attributes, each of these indexes must be updated whenever a new record is inserted. The overhead for updating 10 indexes can slow down the insert operations.

**E. Analyzing the Uniqueness Constraints on Attributes.** Access paths should be specified on all *candidate key* attributes—or sets of attributes—that are either the primary key of a file or unique attributes. The existence of an index (or other access path) makes it sufficient to search only the index when checking this uniqueness constraint, since all values of the attribute will exist in the leaf nodes of the index. For example, when inserting a new record, if a key attribute value of the new record *already exists in the index*, the insertion of the new record should be rejected, since it would violate the uniqueness constraint on the attribute.

Once the preceding information is compiled, it is possible to address the physical database design decisions, which consist mainly of deciding on the storage structures and access paths for the database files.



### 17.7.2 Physical Database Design Decisions

Most relational systems represent each base relation as a physical database file. The access path options include specifying the type of primary file organization for each relation and the attributes that are candidates for defining individual or composite indexes. At most, one of the indexes on each file may be a primary or a clustering index. Any number of additional secondary indexes can be created.

**Design Decisions about Indexing.** The attributes whose values are required in equality or range conditions (selection operation) are those that are keys or that participate in join conditions (join operation) requiring access paths, such as indexes.

The performance of queries largely depends upon what indexes or hashing schemes exist to expedite the processing of selections and joins. On the other hand, during insert, delete, or update operations, the existence of indexes adds to the overhead. This overhead must be justified in terms of the gain in efficiency by expediting queries and transactions.

The physical design decisions for indexing fall into the following categories:

1. **Whether to index an attribute.** The general rules for creating an index on an attribute are that the attribute must either be a key (unique), or there must be some query that uses that attribute either in a selection condition (equality or range of values) or in a join condition. One reason for creating multiple indexes is that some operations can be processed by just scanning the indexes, without having to access the actual data file.
2. **What attribute or attributes to index on.** An index can be constructed on a single attribute, or on more than one attribute if it is a composite index. If multiple attributes from one relation are involved together in several queries, (for example, (Garment\_style\_#, Color) in a garment inventory database), a multiattribute (composite) index is warranted. The ordering of attributes within a multiattribute index must correspond to the queries. For instance, the above index assumes that queries would be based on an ordering of colors within a Garment\_style\_# rather than vice versa.
3. **Whether to set up a clustered index.** At most, one index per table can be a primary or clustering index, because this implies that the file be physically ordered on that attribute. In most RDBMSs, this is specified by the keyword CLUSTER. (If the attribute is a *key*, a *primary index* is created, whereas a *clustering index* is created if the attribute is *not a key*.) If a table requires several indexes, the decision about which one should be the primary or clustering index depends upon whether keeping the table ordered on that attribute is needed. Range queries benefit a great deal from clustering. If several attributes require range queries, relative benefits must be evaluated before deciding which attribute to cluster on. If a query is to be answered by doing an index search only (without retrieving data records), the corresponding index should *not* be clustered, since the main benefit of clustering is achieved



when retrieving the records themselves. A clustering index may be set up as a multiattribute index if range retrieval by that composite key is useful in report creation (for example, an index on `Zip_code`, `Store_id`, and `Product_id` may be a clustering index for sales data).

4. **Whether to use a hash index over a tree index.** In general, RDBMSs use B<sup>+</sup>-trees for indexing. However, ISAM and hash indexes are also provided in some systems. B<sup>+</sup>-trees support both equality and range queries on the attribute used as the search key. Hash indexes work well with equality conditions, particularly during joins to find a matching record(s), but they do not support range queries.
5. **Whether to use dynamic hashing for the file.** For files that are very volatile—that is, those that grow and shrink continuously—one of the dynamic hashing schemes discussed in Section 16.9 would be suitable. Currently, such schemes are not offered by many commercial RDBMSs.

## 17.8 Summary

In this chapter, we presented file organizations that involve additional access structures, called indexes, to improve the efficiency of retrieval of records from a data file. These access structures may be used *in conjunction with* the primary file organizations discussed in Chapter 16, which are used to organize the file records themselves on disk.

Three types of ordered single-level indexes were introduced: primary, clustering, and secondary. Each index is specified on a field of the file. Primary and clustering indexes are constructed on the physical ordering field of a file, whereas secondary indexes are specified on nonordering fields as additional access structures to improve performance of queries and transactions. The field for a primary index must also be a key of the file, whereas it is a nonkey field for a clustering index. A single-level index is an ordered file and is searched using a binary search. We showed how multilevel indexes can be constructed to improve the efficiency of searching an index. An example is IBM's popular indexed sequential access method (ISAM), which is a multilevel index based on the cylinder/track configuration on disk.

Next we showed how multilevel indexes can be implemented as B-trees and B<sup>+</sup>-trees, which are dynamic structures that allow an index to expand and shrink dynamically. The nodes (blocks) of these index structures are kept between half full and completely full by the insertion and deletion algorithms. Nodes eventually stabilize at an average occupancy of 69% full, allowing space for insertions without requiring reorganization of the index for the majority of insertions. B<sup>+</sup>-trees can generally hold more entries in their internal nodes than can B-trees, so they may have fewer levels or hold more entries than does a corresponding B-tree.

We gave an overview of multiple key access methods, and we showed how an index can be constructed based on hash data structures. We introduced the concept of

**partitioned hashing**, which is an extension of external hashing to deal with multiple keys. We also introduced **grid files**, which organize data into buckets along multiple dimensions. We discussed the **hash index** in some detail—it is a secondary structure to access the file by using hashing on a search key other than that used for the primary organization. **Bitmap indexing** is another important type of indexing used for querying by multiple keys and is particularly applicable on fields with a small number of unique values. Bitmaps can also be used at the leaf nodes of B<sup>+</sup> tree indexes as well. We also discussed function-based indexing, which is being provided by relational vendors to allow special indexes on a function of one or more attributes.

We introduced the concept of a logical index and compared it with the physical indexes we described before. They allow an additional level of indirection in indexing in order to permit greater freedom for movement of actual record locations on disk. We discussed index creation in SQL, the process of bulk loading of index files and indexing of strings. We discussed circumstances that point to tuning of indexes. Then we reviewed some general topics related to indexing, including managing constraints, using inverted indexes, and using indexing hints in queries; we commented on column-based storage of relations, which is becoming a viable alternative for storing and accessing large databases. Finally, we discussed physical database design of relational databases, which involves decisions related to storage and accessing of data that we have been discussing in the current and the previous chapter. This discussion was divided into factors that influence the design and the types of decisions regarding whether to index an attribute, what attributes to include in an index, clustered versus nonclustered indexes, hashed indexes, and dynamic hashing.

## Review Questions

- 17.1. Define the following terms: *indexing field*, *primary key field*, *clustering field*, *secondary key field*, *block anchor*, *dense index*, and *nondense (sparse) index*.
- 17.2. What are the differences among primary, secondary, and clustering indexes? How do these differences affect the ways in which these indexes are implemented? Which of the indexes are dense, and which are not?
- 17.3. Why can we have at most one primary or clustering index on a file, but several secondary indexes?
- 17.4. How does multilevel indexing improve the efficiency of searching an index file?
- 17.5. What is the order  $p$  of a B-tree? Describe the structure of B-tree nodes.
- 17.6. What is the order  $p$  of a B<sup>+</sup>-tree? Describe the structure of both internal and leaf nodes of a B<sup>+</sup>-tree.
- 17.7. How does a B-tree differ from a B<sup>+</sup>-tree? Why is a B<sup>+</sup>-tree usually preferred as an access structure to a data file?

- 17.8. Explain what alternative choices exist for accessing a file based on multiple search keys.
- 17.9. What is partitioned hashing? How does it work? What are its limitations?
- 17.10. What is a grid file? What are its advantages and disadvantages?
- 17.11. Show an example of constructing a grid array on two attributes on some file.
- 17.12. What is a fully inverted file? What is an indexed sequential file?
- 17.13. How can hashing be used to construct an index?
- 17.14. What is bitmap indexing? Create a relation with two columns and sixteen tuples and show an example of a bitmap index on one or both.
- 17.15. What is the concept of function-based indexing? What additional purpose does it serve?
- 17.16. What is the difference between a logical index and a physical index?
- 17.17. What is column-based storage of a relational database?

## Exercises

- 17.18. Consider a disk with block size  $B = 512$  bytes. A block pointer is  $P = 6$  bytes long, and a record pointer is  $P_R = 7$  bytes long. A file has  $r = 30,000$  EMPLOYEE records of *fixed length*. Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Department\_code (9 bytes), Address (40 bytes), Phone (10 bytes), Birth\_date (8 bytes), Sex (1 byte), Job\_code (4 bytes), and Salary (4 bytes, real number). An additional byte is used as a deletion marker.
  - a. Calculate the record size  $R$  in bytes.
  - b. Calculate the blocking factor  $bfr$  and the number of file blocks  $b$ , assuming an unspanned organization.
  - c. Suppose that the file is *ordered* by the key field Ssn and we want to construct a *primary index* on Ssn. Calculate (i) the index blocking factor  $bfr_i$  (which is also the index fan-out  $fo$ ); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the primary index.
  - d. Suppose that the file is *not ordered* by the key field Ssn and we want to construct a *secondary index* on Ssn. Repeat the previous exercise (part c) for the secondary index and compare with the primary index.
  - e. Suppose that the file is *not ordered* by the nonkey field Department\_code and we want to construct a *secondary index* on Department\_code, using

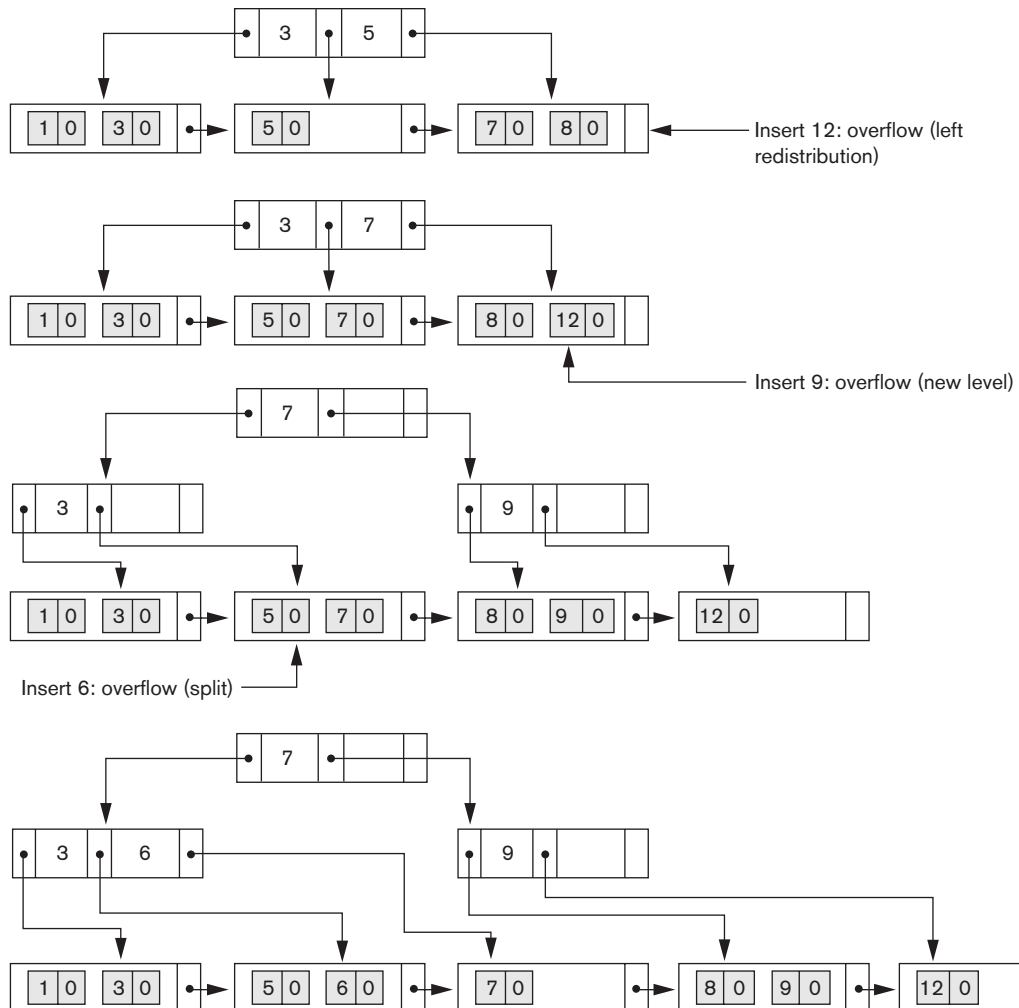
option 3 of Section 17.1.3, with an extra level of indirection that stores record pointers. Assume there are 1,000 distinct values of `Department_code` and that the `EMPLOYEE` records are evenly distributed among these values. Calculate (i) the index blocking factor  $bfr_i$  (which is also the index fan-out  $fo$ ); (ii) the number of blocks needed by the level of indirection that stores record pointers; (iii) the number of first-level index entries and the number of first-level index blocks; (iv) the number of levels needed if we make it into a multilevel index; (v) the total number of blocks required by the multilevel index and the blocks used in the extra level of indirection; and (vi) the approximate number of block accesses needed to search for and retrieve all records in the file that have a specific `Department_code` value, using the index.

- f. Suppose that the file is *ordered* by the nonkey field `Department_code` and we want to construct a *clustering index* on `Department_code` that uses block anchors (every new value of `Department_code` starts at the beginning of a new block). Assume there are 1,000 distinct values of `Department_code` and that the `EMPLOYEE` records are evenly distributed among these values. Calculate (i) the index blocking factor  $bfr_i$  (which is also the index fan-out  $fo$ ); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve all records in the file that have a specific `Department_code` value, using the clustering index (assume that multiple blocks in a cluster are contiguous).
  - g. Suppose that the file is *not* ordered by the key field `Ssn` and we want to construct a  $B^+$ -tree access structure (index) on `Ssn`. Calculate (i) the orders  $p$  and  $p_{leaf}$  of the  $B^+$ -tree; (ii) the number of leaf-level blocks needed if blocks are approximately 69% full (rounded up for convenience); (iii) the number of levels needed if internal nodes are also 69% full (rounded up for convenience); (iv) the total number of blocks required by the  $B^+$ -tree; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its `Ssn` value—using the  $B^+$ -tree.
  - h. Repeat part g, but for a B-tree rather than for a  $B^+$ -tree. Compare your results for the B-tree and for the  $B^+$ -tree.
- 17.19.** A PARTS file with `Part#` as the key field includes records with the following `Part#` values: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suppose that the search field values are inserted in the given order in a  $B^+$ -tree of order  $p = 4$  and  $p_{leaf} = 3$ ; show how the tree will expand and what the final tree will look like.
- 17.20.** Repeat Exercise 17.19, but use a B-tree of order  $p = 4$  instead of a  $B^+$ -tree.
- 17.21.** Suppose that the following search field values are deleted, in the given order, from the  $B^+$ -tree of Exercise 17.19; show how the tree will shrink and show the final tree. The deleted values are 65, 75, 43, 18, 20, 92, 59, 37.

- 17.22.** Repeat Exercise 17.21, but for the B-tree of Exercise 17.20.
- 17.23.** Algorithm 17.1 outlines the procedure for searching a nondense multilevel primary index to retrieve a file record. Adapt the algorithm for each of the following cases:
- A multilevel secondary index on a nonkey nonordering field of a file. Assume that option 3 of Section 17.1.3 is used, where an extra level of indirection stores pointers to the individual records with the corresponding index field value.
  - A multilevel secondary index on a nonordering key field of a file.
  - A multilevel clustering index on a nonkey ordering field of a file.
- 17.24.** Suppose that several secondary indexes exist on nonkey fields of a file, implemented using option 3 of Section 17.1.3; for example, we could have secondary indexes on the fields `Department_code`, `Job_code`, and `Salary` of the `EMPLOYEE` file of Exercise 17.18. Describe an efficient way to search for and retrieve records satisfying a complex selection condition on these fields, such as  $(\text{Department\_code} = 5 \text{ AND } \text{Job\_code} = 12 \text{ AND } \text{Salary} = 50,000)$ , using the record pointers in the indirection level.
- 17.25.** Adapt Algorithms 17.2 and 17.3, which outline search and insertion procedures for a  $B^+$ -tree, to a B-tree.
- 17.26.** It is possible to modify the  $B^+$ -tree insertion algorithm to delay the case where a new level is produced by checking for a possible *redistribution* of values among the leaf nodes. Figure 17.17 illustrates how this could be done for our example in Figure 17.12; rather than splitting the leftmost leaf node when 12 is inserted, we do a *left redistribution* by moving 7 to the leaf node to its left (if there is space in this node). Figure 17.17 shows how the tree would look when redistribution is considered. It is also possible to consider *right redistribution*. Try to modify the  $B^+$ -tree insertion algorithm to take redistribution into account.
- 17.27.** Outline an algorithm for deletion from a  $B^+$ -tree.
- 17.28.** Repeat Exercise 17.27 for a B-tree.

## Selected Bibliography

**Indexing:** Bayer and McCreight (1972) introduced B-trees and associated algorithms. Comer (1979) provides an excellent survey of B-trees and their history, and variations of B-trees. Knuth (1998) provides detailed analysis of many search techniques, including B-trees and some of their variations. Nievergelt (1974) discusses the use of binary search trees for file organization. Textbooks on file structures, including Claybrook (1992), Smith and Barnes (1987), and Salzberg (1988); the algorithms and data structures textbook by Wirth (1985); as well as the database textbook by Ramakrishnan and Gehrke (2003) discuss indexing in detail and may be

**Figure 17.17**B<sup>+</sup>-tree insertion with left redistribution.

consulted for search, insertion, and deletion algorithms for B-trees and B<sup>+</sup>-trees. Larson (1981) analyzes index-sequential files, and Held and Stonebraker (1978) compare static multilevel indexes with B-tree dynamic indexes. Lehman and Yao (1981) and Srinivasan and Carey (1991) did further analysis of concurrent access to B-trees. The books by Wiederhold (1987), Smith and Barnes (1987), and Salzberg (1988), among others, discuss many of the search techniques described in this chapter. Grid files are introduced in Nievergelt et al. (1984). Partial-match retrieval, which uses partitioned hashing, is discussed in Burkhard (1976, 1979).

New techniques and applications of indexes and B<sup>+</sup>-trees are discussed in Lanka and Mays (1991), Zobel et al. (1992), and Faloutsos and Jagadish (1992). Mohan

and Narang (1992) discuss index creation. The performance of various B-tree and B<sup>+</sup>-tree algorithms is assessed in Baeza-Yates and Larson (1989) and Johnson and Shasha (1993). Buffer management for indexes is discussed in Chan et al. (1992). Column-based storage of databases was proposed by Stonebraker et al. (2005) in the C-Store database system; MonetDB/X100 by Boncz et al. (2008) is another implementation of the idea. Abadi et al. (2008) discuss the advantages of column stores over row-stored databases for read-only database applications.

**Physical Database Design:** Wiederhold (1987) covers issues related to physical design. O'Neil and O'Neil (2001) provides a detailed discussion of physical design and transaction issues in reference to commercial RDBMSs. Navathe and Kerschberg (1986) discuss all phases of database design and point out the role of data dictionaries. Rozen and Shasha (1991) and Carlis and March (1984) present different models for the problem of physical database design. Shasha and Bonnet (2002) offer an elaborate discussion of guidelines for database tuning. Niemiec (2008) is one among several books available for Oracle database administration and tuning; Schneider (2006) is focused on designing and tuning MySQL databases.

# part 8

## **Query Processing and Optimization**



This page intentionally left blank

## Strategies for Query Processing<sup>1</sup>

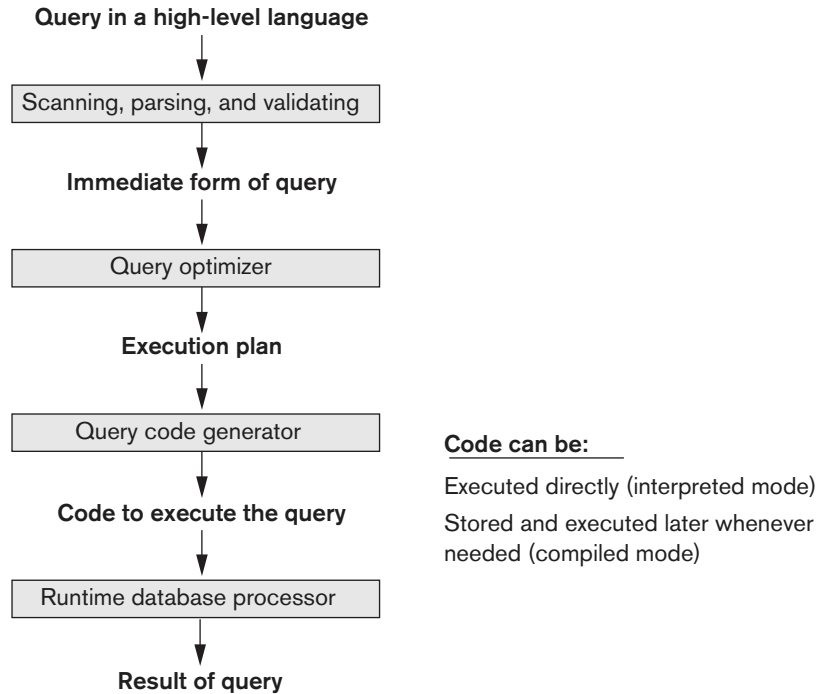
In this chapter, we discuss the techniques used internally by a DBMS to process high-level queries. A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.<sup>2</sup> The **scanner** identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query, whereas the **parser** checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language. The query must also be **validated** by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried. An internal representation of the query is then created, usually as a tree data structure called a **query tree**. It is also possible to represent the query using a graph data structure called a **query graph**, which is generally a **directed acyclic graph (DAG)**. The DBMS must then devise an **execution strategy** or **query plan** for retrieving the results of the query from the database files. A query has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.

We defer a detailed discussion of query optimization to the next chapter. In this chapter, we will primarily focus on how queries are processed and what algorithms are used to perform individual operations within the query. Figure 18.1 shows the different steps of processing a high-level query. The **query optimizer** module has the task of producing a good execution plan, and the **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running (executing) the query code, whether in compiled or interpreted mode, to produce the query result. If a runtime error results, an error message is generated by the runtime database processor.

---

<sup>1</sup>We appreciate Rafi Ahmed's contributions in updating this chapter.

<sup>2</sup>We will not discuss the parsing and syntax-checking phase of query processing here; this material is discussed in compiler texts.



**Figure 18.1**  
Typical steps when  
processing a high-level  
query.

The term *optimization* is actually a misnomer because in some cases the chosen execution plan is not the optimal (or absolute best) strategy—it is just a *reasonably efficient or the best available strategy* for executing the query. Finding the optimal strategy is usually too time-consuming—except for the simplest of queries. In addition, trying to find the optimal query execution strategy requires accurate and detailed information about the size of the tables and distributions of things such as column values, which may not be always available in the DBMS catalog. Furthermore, additional information such as the size of the expected result must be derived based on the predicates in the query. Hence, *planning of a good execution strategy* may be a more accurate description than *query optimization*.

For lower-level navigational database languages in legacy systems—such as the network DML or the hierarchical DL/1 the programmer must choose the query execution strategy while writing a database program. If a DBMS provides only a navigational language, there is a *limited opportunity* for extensive query optimization by the DBMS; instead, the programmer is given the capability to choose the query execution strategy. On the other hand, a high-level query language—such as SQL for relational DBMSs (RDBMSs) or OQL (see Chapter 12) for object DBMSs (ODBMSs)—is more declarative in nature because it specifies what the intended results of the query are rather than identifying the details of *how* the result should be obtained. Query optimization is thus necessary for queries that are specified in a high-level query language.

We will concentrate on describing query processing and optimization in the *context of an RDBMS* because many of the techniques we describe have also been adapted for other types of database management systems, such as ODBMSs.<sup>3</sup> A relational DBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or near-optimal strategy. Most DBMSs have a number of general database access algorithms that implement relational algebra operations such as SELECT or JOIN (see Chapter 8) or combinations of these operations. Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query, as well as to the *particular physical database design*, can be considered by the query optimization module.

This chapter is organized as follows. Section 18.1 starts with a general discussion of how SQL queries are typically translated into relational algebra queries and additional operations and then optimized. Then we discuss algorithms for implementing relational algebra operations in Sections 18.2 through 18.6. In Section 18.7, we discuss the strategy for execution called pipelining. Section 18.8 briefly reviews the strategy for parallel execution of the operators. Section 18.9 summarizes the chapter.

In the next chapter, we will give an overview of query optimization strategies. There are two main techniques of query optimization that we will be discussing. The first technique is based on **heuristic rules** for ordering the operations in a query execution strategy that works well in most cases but is not guaranteed to work well in every case. The rules typically reorder the operations in a query tree. The second technique involves **cost estimation** of different execution strategies and choosing the execution plan that minimizes estimated cost. The topics covered in this chapter require that the reader be familiar with the material presented in several earlier chapters. In particular, the chapters on SQL (Chapters 6 and 7), relational algebra (Chapter 8), and file structures and indexing (Chapters 16 and 17) are a prerequisite to this chapter. Also, it is important to note that the topic of query processing and optimization is vast, and we can only give an introduction to the basic principles and techniques in this and the next chapter. Several important works are mentioned in the Bibliography of this and the next chapter.

## 18.1 Translating SQL Queries into Relational Algebra and Other Operators

In practice, SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into *query blocks*, which form the basic units that can be translated into the algebraic operators and optimized. A **query block** contains a single SELECT-FROM-WHERE expression, as well as GROUP BY

---

<sup>3</sup>There are some query processing and optimization issues and techniques that are pertinent only to ODBMSs. However, we do not discuss them here because we give only an introduction to query processing in this chapter and we do not discuss query optimization until Chapter 19.

and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, and COUNT—these operators must also be included in the extended algebra, as we discussed in Section 8.4.

Consider the following SQL query on the EMPLOYEE relation in Figure 5.5:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ( SELECT MAX (Salary)
                  FROM EMPLOYEE
                  WHERE Dno=5 );
```

This query retrieves the names of employees (from any department in the company) who earn a salary that is greater than the *highest salary in department 5*. The query includes a nested subquery and hence would be decomposed into two blocks. The inner block is:

```
( SELECT MAX (Salary)
  FROM EMPLOYEE
  WHERE Dno=5 )
```

This retrieves the highest salary in department 5. The outer query block is:

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > c
```

where  $c$  represents the result returned from the inner block. The inner block could be translated into the following extended relational algebra expression:

$$\mathcal{S}_{\text{MAX Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$$

and the outer block into the expression:

$$\pi_{\text{Lname, Fname}}(\sigma_{\text{Salary} > c}(\text{EMPLOYEE}))$$

The *query optimizer* would then choose an execution plan for each query block. Notice that in the above example, the inner block needs to be evaluated only once to produce the maximum salary of employees in department 5, which is then used—as the constant  $c$ —by the outer block. We called this a *nested subquery block (which is uncorrelated to the outer query block)* in Section 7.1.2. It is more involved to optimize the more complex *correlated nested subqueries* (see Section 7.1.3), where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block. Many techniques are used in advanced DBMSs to unnest and optimize correlated nested subqueries.

### 18.1.1 Additional Operators Semi-Join and Anti-Join

Most RDBMSs currently process SQL queries arising from various types of enterprise applications that include ad hoc queries, standard canned queries with parameters,

and queries for report generation. Additionally, SQL queries originate from OLAP (online analytical processing) applications on data warehouses (we discuss data warehousing in detail in Chapter 29). Some of these queries are transformed into operations that are not part of the standard relational algebra we discussed in Chapter 8. Two commonly used operations are **semi-join** and **anti-join**. Note that both these operations are a type of join. Semi-join is generally used for unnesting EXISTS, IN, and ANY subqueries.<sup>4</sup> Here we represent semi-join by the following non-standard syntax:  $T1.X \text{ } S = T2.Y$ , where  $T1$  is the left table and  $T2$  is the right table of the semi-join. The semantics of semi-join are as follows: A row of  $T1$  is returned as soon as  $T1.X$  finds a match with any value of  $T2.Y$  without searching for further matches. This is in contrast to finding all possible matches in inner join.

Consider a slightly modified version of the schema in Figure 5.5 as follows:

```
EMPLOYEE ( Ssn, Bdate, Address, Sex, Salary, Dno)
DEPARTMENT ( Dnumber, Dname, Dmgrssn, Zipcode)
```

where a department is located in a specific zip code.

Let us consider the following query:

```
Q (S) : SELECT COUNT(*)
FROM   DEPARTMENT D
WHERE  D.Dnumber IN ( SELECT  E.Dno
                      FROM    EMPLOYEE E
                      WHERE    E.Salary > 200000)
```

Here we have a nested query which is joined by the connector **IN**.

To remove the nested query:

```
( SELECT  E.Dno
  FROM    EMPLOYEE E  WHERE E.Salary > 200000)
```

is called as **unnesting**. It leads to the following query with an operation called **semi-join**,<sup>5</sup> which we show with a non-standard notation “ $S=$ ” below:

```
SELECT COUNT(*)
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  D.Dnumber S= E.Dno and E.Salary > 200000;
```

The above query is counting the number of departments that have employees who make more than \$200,000 annually. Here, the operation is to find the department whose *Dnumber* attribute matches the value(s) for the *Dno* attribute of Employee with that high salary.

<sup>4</sup>In some cases where duplicate rows are not relevant, inner join can also be used to unnest EXISTS and ANY subqueries.

<sup>5</sup>Note that this semi-join operator is not the same as that used in distributed query processing.

In algebra, alternate notations exist. One common notation is shown in the following figure.

Semi-join



Now consider another query:

```
Q (AJ) : SELECT COUNT(*)
FROM   EMPLOYEE
WHERE  EMPLOYEE.Dno NOT IN (SELECT DEPARTMENT.Dnumber
                                FROM   DEPARTMENT
                                WHERE  Zipcode =30332)
```

The above query counts the number of employees who *do not* work in departments located in zip code 30332. Here, the operation is to find the employee tuples whose Dno attribute does *not* match the value(s) for the Dnumber attribute in DEPARTMENT for the given zip code. We are only interested in producing a count of such employees, and performing an inner join of the two tables would, of course, produce wrong results. In this case, therefore, the **anti-join** operator is used while unnesting this query.

Anti-join is used for unnesting NOT EXISTS, NOT IN, and ALL subqueries. We represent anti-join by the following nonstandard syntax:  $T1.x \text{ A } T2.y$ , where  $T1$  is the left table and  $T2$  is the right table of the anti-join. The semantics of anti-join are as follows: A row of  $T1$  is rejected as soon as  $T1.x$  finds a match with any value of  $T2.y$ . A row of  $T1$  is returned, only if  $T1.x$  does not match with any value of  $T2.y$ .

In the following result of unnesting, we show the aforementioned anti-join with the nonstandard symbol “A=” in the following:

```
SELECT COUNT(*)
FROM   EMPLOYEE, DEPARTMENT
WHERE  EMPLOYEE.Dno A= DEPARTMENT AND Zipcode =30332
```

In algebra, alternate notations exist. One common notation is shown in the following figure.

Anti-join



## 18.2 Algorithms for External Sorting

Sorting is one of the primary algorithms used in query processing. For example, whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted. Sorting is also a key component in sort-merge algorithms used for JOIN and

other operations (such as UNION and INTERSECTION), and in duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT option in the SELECT clause). We will discuss one of these algorithms in this section. Note that sorting of a particular file may be avoided if an appropriate index—such as a primary or clustering index (see Chapter 17)—exists on the desired file attribute to allow ordered access to the records of the file.

**External sorting** refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.<sup>6</sup> The typical external sorting algorithm uses a **sort-merge strategy**, which starts by sorting small subfiles—called **runs**—of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn. The sort-merge algorithm, like other database algorithms, requires *buffer space* in main memory, where the actual sorting and merging of the runs is performed. The basic algorithm, outlined in Figure 18.2, consists of two phases: the sorting phase and the merging phase. The buffer space in main memory is part of the **DBMS cache**—an area in the computer's main memory that is controlled by the DBMS. The buffer space is divided into individual buffers, where each **buffer** is the same size in bytes as the size of one disk block. Thus, one buffer can hold the contents of exactly *one disk block*.

In the **sorting phase**, runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an *internal* sorting algorithm, and written back to disk as temporary sorted subfiles (or runs). The size of each run and the **number of initial runs** ( $n_R$ ) are dictated by the **number of file blocks** ( $b$ ) and the **available buffer space** ( $n_B$ ). For example, if the number of available main memory buffers  $n_B = 5$  disk blocks and the size of the file  $b = 1,024$  disk blocks, then  $n_R = \lceil (b/n_B) \rceil$  or 205 initial runs each of size 5 blocks (except the last run, which will have only 4 blocks). Hence, after the sorting phase, 205 sorted runs (or 205 sorted subfiles of the original file) are stored as temporary subfiles on disk.

In the **merging phase**, the sorted runs are merged during one or more **merge passes**. Each merge pass can have one or more merge steps. The **degree of merging** ( $d_M$ ) is the number of sorted subfiles that can be merged in each merge step. During each merge step, one buffer block is needed to hold one disk block from each of the sorted subfiles being merged, and one additional buffer is needed for containing one disk block of the merge result, which will produce a larger sorted file that is the result of merging several smaller sorted subfiles. Hence,  $d_M$  is the smaller of  $(n_B - 1)$  and  $n_R$ , and the number of merge passes is  $\lceil (\log_{d_M}(n_R)) \rceil$ . In our example, where  $n_B = 5$ ,  $d_M = 4$  (four-way merging), so the 205 initial sorted runs would be merged 4 at a time in each step into 52 larger sorted subfiles at the end of the first merge pass. These 52 sorted files are then merged 4 at a time into 13 sorted files, which are then merged into 4 sorted files, and then finally into 1 fully sorted file, which means that *four passes* are needed.

---

<sup>6</sup>*Internal sorting algorithms* are suitable for sorting data structures, such as tables and lists, that can fit entirely in main memory. These algorithms are described in detail in data structures and algorithms texts, and include techniques such as quick sort, heap sort, bubble sort, and many others. We do not discuss these here. Also, main-memory DBMSs such as HANA employ their own techniques for sorting.



```

set     $i \leftarrow 1$ ;
         $j \leftarrow b$ ;           {size of the file in blocks}
         $k \leftarrow n_B$ ;         {size of buffer in blocks}
         $m \leftarrow \lceil (j/k) \rceil$ ; {number of subfiles- each fits in buffer}
{Sorting Phase}
while ( $i \leq m$ )
do {
    read next  $k$  blocks of the file into the buffer or if there are less than  $k$  blocks
        remaining, then read in the remaining blocks;
    sort the records in the buffer and write as a temporary subfile;
     $i \leftarrow i + 1$ ;
}
{Merging Phase: merge subfiles until only 1 remains}
set     $i \leftarrow 1$ ;
         $p \leftarrow \lceil \log_{k-1} m \rceil$  { $p$  is the number of passes for the merging phase}
         $j \leftarrow m$ ;
while ( $i \leq p$ )
do {
     $n \leftarrow 1$ ;
     $q \leftarrow \lceil j/(k-1) \rceil$ ; {number of subfiles to write in this pass}
    while ( $n \leq q$ )
    do {
        read next  $k-1$  subfiles or remaining subfiles (from previous pass)
            one block at a time;
        merge and write as new subfile one block at a time;
         $n \leftarrow n + 1$ ;
    }
     $j \leftarrow q$ ;
     $i \leftarrow i + 1$ ;
}

```

**Figure 18.2**  
Outline of the  
sort-merge  
algorithm for  
external sorting.

The performance of the sort-merge algorithm can be measured in terms of the number of disk block reads and writes (between the disk and main memory) before the sorting of the whole file is completed. The following formula approximates this cost:

$$(2 * b) + (2 * b * (\log_{dM} n_R))$$

The first term  $(2 * b)$  represents the number of block accesses for the sorting phase, since each file block is accessed twice: once for reading into a main memory buffer and once for writing the sorted records back to disk into one of the sorted subfiles. The second term represents the number of block accesses for the merging phase. During each merge pass, a number of disk blocks approximately equal to the original file blocks  $b$  is read and written. Since the number of merge passes is  $(\log_{dM} n_R)$ , we get the total merge cost of  $(2 * b * (\log_{dM} n_R))$ .

The minimum number of main memory buffers needed is  $n_B = 3$ , which gives a  $d_M$  of 2 and an  $n_R$  of  $\lceil (b/3) \rceil$ . The minimum  $d_M$  of 2 gives the worst-case performance of the algorithm, which is:

$$(2 * b) + (2 * (b * (\log_2 n_R))).$$

The following sections discuss the various algorithms for the operations of the relational algebra (see Chapter 8).

## 18.3 Algorithms for SELECT Operation

### 18.3.1 Implementation Options for the SELECT Operation

There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition. Some of the search algorithms depend on the file having specific access paths, and they may apply only to certain types of selection conditions. We discuss some of the algorithms for implementing SELECT in this section. We will use the following operations, specified on the relational database in Figure 5.5, to illustrate our discussion:

OP1:  $\sigma_{\text{Ssn} = '123456789'}$  (EMPLOYEE)  
 OP2:  $\sigma_{\text{Dnumber} > 5}$  (DEPARTMENT)  
 OP3:  $\sigma_{\text{Dno} = 5}$  (EMPLOYEE)  
 OP4:  $\sigma_{\text{Dno} = 5 \text{ AND Salary} > 30000 \text{ AND Sex} = 'F'}$  (EMPLOYEE)  
 OP5:  $\sigma_{\text{Essn} = '123456789' \text{ AND Pno} = 10}$  (WORKS\_ON)  
 OP6: An SQL Query:  
     **SELECT** \*  
     **FROM** EMPLOYEE  
     **WHERE** Dno IN (3,27, 49)

OP7: An SQL Query (from Section 17.5.3)  
     **SELECT** First\_name, Lname  
     **FROM** Employee  
     **WHERE** ((Salary\*Commission\_pct) + Salary ) > 15000;

**Search Methods for Simple Selection.** A number of search algorithms are possible for selecting records from a file. These are also known as **file scans**, because they scan the records of a file to search for and retrieve records that satisfy a selection condition.<sup>7</sup> If the search algorithm involves the use of an index, the index search is called an **index scan**. The following search methods (S1 through S6) are examples of some of the search algorithms that can be used to implement a select operation:

- **S1—Linear search (brute force algorithm).** Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition. Since the

<sup>7</sup>A selection operation is sometimes called a **filter**, since it filters out the records in the file that do *not* satisfy the selection condition.

records are grouped into disk blocks, each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

- **S2—Binary search.** If the selection condition involves an equality comparison on a key attribute on which the file is **ordered**, binary search—which is more efficient than linear search—can be used. An example is OP1 if Ssn is the ordering attribute for the EMPLOYEE file.<sup>8</sup>
- **S3a—Using a primary index.** If the selection condition involves an equality comparison on a **key attribute** with a primary index—for example, Ssn = '123456789' in OP1—use the primary index to retrieve the record. Note that this condition retrieves a single record (at most).
- **S3b—Using a hash key.** If the selection condition involves an equality comparison on a **key attribute** with a hash key—for example, Ssn = '123456789' in OP1—use the hash key to retrieve the record. Note that this condition retrieves a single record (at most).
- **S4—Using a primary index to retrieve multiple records.** If the comparison condition is >, >=, <, or <= on a key field with a primary index—for example, Dnumber > 5 in OP2—use the index to find the record satisfying the corresponding equality condition (Dnumber = 5); then retrieve all subsequent records in the (ordered) file. For the condition Dnumber < 5, retrieve all the preceding records.
- **S5—Using a clustering index to retrieve multiple records.** If the selection condition involves an equality comparison on a **nonkey attribute** with a clustering index—for example, Dno = 5 in OP3—use the index to retrieve all the records satisfying the condition.
- **S6—Using a secondary (B<sup>+</sup>-tree) index on an equality comparison.** This search method can be used to retrieve a single record if the indexing field is a **key** (has unique values) or to retrieve multiple records if the indexing field is **not a key**. This can also be used for comparisons involving >, >=, <, or <=. Queries involving a range of values (e.g., 3,000 <= Salary <= 4,000) in their selection are called **range queries**. In case of range queries, the B<sup>+</sup>-tree index leaf nodes contain the indexing field value in order—so a sequence of them is used corresponding to the requested range of that field and provide record pointers to the qualifying records.
- **S7a—Using a bitmap index.** (See Section 17.5.2.) If the selection condition involves a set of values for an attribute (e.g., Dnumber in (3,27,49) in OP6), the corresponding bitmaps for each value can be OR-ed to give the set of record ids that qualify. In this example, that amounts to OR-ing three bitmap vectors whose length is the same as the number of employees.

---

<sup>8</sup>Generally, binary search is not used in database searches because ordered files are not used unless they also have a corresponding primary index.

- **S7b—Using a functional index.** (See Section 17.5.3.) In OP7, the selection condition involves the expression  $((\text{Salary} * \text{Commission\_pct}) + \text{Salary})$ . If there is a functional index defined as (as shown in Section 17.5.3):

```
CREATE INDEX income_ix
ON EMPLOYEE (Salary + (Salary*Commission_pct));
```

then this index can be used to retrieve employee records that qualify. Note that the exact way in which the function is written while creating the index is immaterial.

In the next chapter, we discuss how to develop formulas that estimate the access cost of these search methods in terms of the number of block accesses and access time. Method S1 (**linear search**) applies to any file, but all the other methods depend on having the appropriate access path on the attribute used in the selection condition. Method S2 (**binary search**) requires the file to be sorted on the search attribute. The methods that use an index (S3a, S4, S5, and S6) are generally referred to as **index searches**, and they require the appropriate index to exist on the search attribute. Methods S4 and S6 can be used to retrieve records in a certain *range* in **range queries**. Method S7a (**bitmap index search**) is suitable for retrievals where an attribute must match an enumerated set of values. Method S7b (**functional index search**) is suitable when the match is based on a function of one or more attributes on which a functional index exists.

### 18.3.2 Search Methods for Conjunctive Selection

If a condition of a SELECT operation is a **conjunctive condition**—that is, if it is made up of several simple conditions connected with the AND logical connective such as OP4 above—the DBMS can use the following additional methods to implement the operation:

- **S8—Conjunctive selection using an individual index.** If an attribute involved in any **single simple condition** in the conjunctive select condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record *satisfies the remaining simple conditions* in the conjunctive select condition.
- **S9—Conjunctive selection using a composite index.** If two or more attributes are involved in equality conditions in the conjunctive select condition and a composite index (or hash structure) exists on the combined fields—for example, if an index has been created on the composite key (Essn, Pno) of the WORKS\_ON file for OP5—we can use the index directly.
- **S10—Conjunctive selection by intersection of record pointers.**<sup>9</sup> If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if

<sup>9</sup>A record pointer uniquely identifies a record and provides the address of the record on disk; hence, it is also called the **record identifier** or **record id**.

the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the **set of record pointers** that satisfy the individual condition. The **intersection** of these sets of record pointers gives the record pointers that satisfy the conjunctive select condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.<sup>10</sup> In general, method S10 assumes that each of the indexes is on a *nonkey field* of the file, because if one of the conditions is an equality condition on a key field, only one record will satisfy the whole condition. The bitmap and functional indexes discussed above in S7 are applicable for conjunctive selection on multiple attributes as well. For conjunctive selection on multiple attributes, the resulting bitmaps are AND-ed to produce the list of record ids; the same can be done when one or more set of record ids comes from a functional index.

Whenever a single condition specifies the selection—such as OP1, OP2, or OP3—the DBMS can only check whether or not an access path exists on the attribute involved in that condition. If an access path (such as index or hash key or bitmap index or sorted file) exists, the method corresponding to that access path is used; otherwise, the brute force, linear search approach of method S1 can be used. Query optimization for a SELECT operation is needed mostly for conjunctive select conditions whenever *more than one* of the attributes involved in the conditions have an access path. The optimizer should choose the access path that *retrieves the fewest records* in the most efficient way by estimating the different costs (see Section 19.3) and choosing the method with the least estimated cost.

### 18.3.3 Search Methods for Disjunctive Selection

Compared to a conjunctive selection condition, a **disjunctive condition** (where simple conditions are connected by the OR logical connective rather than by AND) is much harder to process and optimize. For example, consider OP4':

OP4':  $\sigma_{Dno=5 \text{ OR Salary} > 30000 \text{ OR Sex} = 'F'}(\text{EMPLOYEE})$

With such a condition, the records satisfying the disjunctive condition are the *union* of the records satisfying the individual conditions. Hence, if any *one* of the conditions does not have an access path, we are compelled to use the brute force, linear search approach. Only if an access path exists on *every* simple condition in the disjunction can we optimize the selection by retrieving the records satisfying each condition—or their record ids—and then applying the *union* operation to eliminate duplicates.

All the methods discussed in S1 through S7 are applicable for each simple condition yielding a possible set of record ids. The query optimizer must choose the appropriate one for executing each SELECT operation in a query. This optimization uses

<sup>10</sup>The technique can have many variations—for example, if the indexes are *logical indexes* that store primary key values instead of record pointers.

formulas that estimate the costs for each available access method, as we will discuss in Sections 19.4 and 19.5. The optimizer chooses the access method with the lowest estimated cost.

### 18.3.4 Estimating the Selectivity of a Condition

To minimize the overall cost of query execution in terms of resources used and response time, the query optimizer receives valuable input from the system catalog, which contains crucial statistical information about the database.

**Information in the Database Catalog.** A typical RDBMS catalog contains the following types of information:

For each relation (table)  $r$  with schema  $R$  containing  $r_R$  tuples:

- The number of rows/records or its cardinality:  $|r(R)|$ . We will refer to the number of rows simply as  $r_R$ .
- The “width” of the relation (i.e., the length of each tuple in the relation) this length of tuple is referred to as  $R$ .
- The number of blocks that relation occupies in storage: referred to as  $b_R$ .
- The blocking factor  $bfr$ , which is the number of tuples per block.

For each attribute  $A$  in relation  $R$ :

- The number of distinct values of  $A$  in  $R$ :  $NDV(A, R)$ .
- The max and min values of attribute  $A$  in  $R$ :  $\max(A, R)$  and  $\min(A, R)$ .

Note that many other forms of the statistics are possible and may be kept as needed. If there is a composite index on attributes  $\langle A, B \rangle$ , then the  $NDV(R, \langle A, B \rangle)$  is of significance. An effort is made to keep these statistics as accurate as possible; however, keeping them accurate up-to-the-minute is considered unnecessary since the overhead of doing so in fairly active databases is too high. We will be revisiting many of the above parameters again in Section 19.3.2.

When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the *selectivity* of each condition. The **selectivity ( $sI$ )** is defined as the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation), and thus it is a number between zero and one. *Zero selectivity* means none of the records in the file satisfies the selection condition, and a selectivity of one means that all the records in the file satisfy the condition. In general, the selectivity will not be either of these two extremes, but will be a fraction that estimates the percentage of file records that will be retrieved.

Although exact selectivities of all conditions may not be available, **estimates of selectivities** are possible from the information kept in the DBMS catalog and are used by the optimizer. For example, for an equality condition on a key attribute of relation  $r(R)$ ,  $s = 1/|r(R)|$ , where  $|r(R)|$  is the number of tuples in relation  $r(R)$ . For an equality condition on a nonkey attribute with  $i$  *distinct values*,  $s$  can be estimated by

$(|r(R)|/i)/|r(R)|$  or  $1/i$ , assuming that the records are evenly or **uniformly distributed** among the distinct values. Under this assumption,  $|r(R)|/i$  records will satisfy an equality condition on this attribute. For a range query with the selection condition,

$$\begin{aligned} A &\geq v, \text{ assuming uniform distribution,} \\ sl &= 0 \text{ if } v > \max(A, R) \\ sl &= \max(A, R) - v / \max(A, R) - \min(A, R) \end{aligned}$$

In general, the number of records satisfying a selection condition with selectivity  $sl$  is estimated to be  $|r(R)| * sl$ . The smaller this estimate is, the higher the desirability of using that condition first to retrieve records. For a nonkey attribute with NDV  $(A, R)$  distinct values, it is often the case that those values are not uniformly distributed.

If the actual distribution of records among the various distinct values of the attribute is kept by the DBMS in the form of a **histogram**, it is possible to get more accurate estimates of the number of records that satisfy a particular condition. We will discuss the catalog information and histograms in more detail in Section 19.3.3.

## 18.4 Implementing the JOIN Operation

The JOIN operation is one of the most time-consuming operations in query processing. Many of the join operations encountered in queries are of the EQUIJOIN and NATURAL JOIN varieties, so we consider just these two here since we are only giving an overview of query processing and optimization. For the remainder of this chapter, the term **join** refers to an EQUIJOIN (or NATURAL JOIN).

There are many possible ways to implement a **two-way join**, which is a join on two files. Joins involving more than two files are called **multiway joins**. The number of possible ways to execute multiway joins grows rapidly because of the combinatorial explosion of possible join orderings. In this section, we discuss techniques for implementing *only two-way joins*. To illustrate our discussion, we refer to the relational schema shown in Figure 5.5 once more—specifically, to the EMPLOYEE, DEPARTMENT, and PROJECT relations. The algorithms we discuss next are for a join operation of the form:

$$R \bowtie_{A=B} S$$

where  $A$  and  $B$  are the **join attributes**, which should be domain-compatible attributes of  $R$  and  $S$ , respectively. The methods we discuss can be extended to more general forms of join. We illustrate four of the most common techniques for performing such a join, using the following sample operations:

OP6: EMPLOYEE  $\bowtie_{Dno=Dnumber}$  DEPARTMENT  
 OP7: DEPARTMENT  $\bowtie_{Mgr\_ssn=Ssn}$  EMPLOYEE

### 18.4.1 Methods for Implementing Joins

- **J1—Nested-loop join (or nested-block join).** This is the default (brute force) algorithm because it does not require any special access paths on either file in the



join. For each record  $t$  in  $R$  (outer loop), retrieve every record  $s$  from  $S$  (inner loop) and test whether the two records satisfy the join condition  $t[A] = s[B]$ .<sup>11</sup>

- **J2—Index-based nested-loop join (using an access structure to retrieve the matching records).** If an index (or hash key) exists for one of the two join attributes—say, attribute  $B$  of file  $S$ —retrieve each record  $t$  in  $R$  (loop over file  $R$ ), and then use the access structure (such as an index or a hash key) to retrieve directly all matching records  $s$  from  $S$  that satisfy  $s[B] = t[A]$ .
- **J3—Sort-merge join.** If the records of  $R$  and  $S$  are *physically sorted* (ordered) by value of the join attributes  $A$  and  $B$ , respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for  $A$  and  $B$ . If the files are not sorted, they may be sorted first by using external sorting (see Section 18.2). In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file—unless both  $A$  and  $B$  are nonkey attributes, in which case the method needs to be modified slightly. A sketch of the sort-merge join algorithm is given in Figure 18.3(a). We use  $R(i)$  to refer to the  $i$ th record in file  $R$ . A variation of the sort-merge join can be used when secondary indexes exist on both join attributes. The indexes provide the ability to access (scan) the records in order of the join attributes, but the records themselves are physically scattered all over the file blocks, so this method may be inefficient because every record access may involve accessing a different disk block.
- **J4—Partition-hash join (or just hash-join).** The records of files  $R$  and  $S$  are partitioned into smaller files. The partitioning of each file is done using the same hashing function  $h$  on the join attribute  $A$  of  $R$  (for partitioning file  $R$ ) and  $B$  of  $S$  (for partitioning file  $S$ ). First, a single pass through the file with fewer records (say,  $R$ ) hashes its records to the various partitions of  $R$ ; this is called the **partitioning phase**, since the records of  $R$  are partitioned into the hash buckets. In the simplest case, we assume that the smaller file can fit entirely in main memory after it is partitioned, so that the partitioned subfiles of  $R$  are all kept in main memory. The collection of records with the same value of  $h(A)$  are placed in the same partition, which is a **hash bucket** in a hash table in main memory. In the second phase, called the **probing phase**, a single pass through the other file ( $S$ ) then hashes each of its records using the same hash function  $h(B)$  to *probe* the appropriate bucket, and that record is combined with all matching records from  $R$  in that bucket. This simplified description of partition-hash join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase. We will discuss the general case of partition-hash join below that does not require this assumption. In practice, techniques J1 to J4 are implemented by accessing *whole disk blocks* of a file, rather than individual records. Depending on the available number of buffers in memory, the number of blocks read in from the file can be adjusted.

<sup>11</sup>For disk files, it is obvious that the loops will be over disk blocks, so this technique has also been called *nested-block join*.



**Figure 18.3**

Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where  $R$  has  $n$  tuples and  $S$  has  $m$  tuples. (a) Implementing the operation  $T \leftarrow R \bowtie_{A=B} S$ . (b) Implementing the operation  $T \leftarrow \pi_{\langle \text{attribute list} \rangle}(R)$ .

```

(a) sort the tuples in  $R$  on attribute  $A$ ;                                (*assume  $R$  has  $n$  tuples (records)*)
    sort the tuples in  $S$  on attribute  $B$ ;                                (*assume  $S$  has  $m$  tuples (records)*)
    set  $i \leftarrow 1, j \leftarrow 1$ ;
    while  $(i \leq n)$  and  $(j \leq m)$ 
    do { if  $R(i)[A] > S(j)[B]$ 
        then set  $j \leftarrow j + 1$ 
        elseif  $R(i)[A] < S(j)[B]$ 
        then set  $i \leftarrow i + 1$ 
        else { (*  $R(i)[A] = S(j)[B]$ , so we output a matched tuple *)
            output the combined tuple  $\langle R(i), S(j) \rangle$  to  $T$ ;

            (* output other tuples that match  $R(i)$ , if any *)
            set  $l \leftarrow j + 1$ ;
            while  $(l \leq m)$  and  $(R(i)[A] = S(l)[B])$ 
            do { output the combined tuple  $\langle R(i), S(l) \rangle$  to  $T$ ;
                set  $l \leftarrow l + 1$ 
            }

            (* output other tuples that match  $S(j)$ , if any *)
            set  $k \leftarrow i + 1$ ;
            while  $(k \leq n)$  and  $(R(k)[A] = S(j)[B])$ 
            do { output the combined tuple  $\langle R(k), S(j) \rangle$  to  $T$ ;
                set  $k \leftarrow k + 1$ 
            }
            set  $i \leftarrow k, j \leftarrow l$ 
        }
    }

(b) create a tuple  $t[\langle \text{attribute list} \rangle]$  in  $T'$  for each tuple  $t$  in  $R$ ;
    (*  $T'$  contains the projection results before duplicate elimination *)
    if  $\langle \text{attribute list} \rangle$  includes a key of  $R$ 
    then  $T \leftarrow T'$ 
    else { sort the tuples in  $T'$ ;
        set  $i \leftarrow 1, j \leftarrow 2$ ;
        while  $i \leq n$ 
        do { output the tuple  $T'[i]$  to  $T$ ;
            while  $T'[i] = T'[j]$  and  $j \leq n$  do  $j \leftarrow j + 1$ ;          (* eliminate duplicates *)
             $i \leftarrow j; j \leftarrow i + 1$ 
        }
    }
    (*  $T$  contains the projection result after duplicate elimination *)

```

**Figure 18.3 (continued)**

Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where  $R$  has  $n$  tuples and  $S$  has  $m$  tuples. (c) Implementing the operation  $T \leftarrow R \cup S$ . (d) Implementing the operation  $T \leftarrow R \cap S$ . (e) Implementing the operation  $T \leftarrow R - S$ .

- (c) sort the tuples in  $R$  and  $S$  using the same unique sort attributes;  
 set  $i \leftarrow 1, j \leftarrow 1$ ;  
 while  $(i \leq n)$  and  $(j \leq m)$   
 do { if  $R(i) > S(j)$   
     then { output  $S(j)$  to  $T$ ;  
         set  $j \leftarrow j + 1$   
     }  
     elseif  $R(i) < S(j)$   
     then { output  $R(i)$  to  $T$ ;  
         set  $i \leftarrow i + 1$   
     }  
     else set  $j \leftarrow j + 1$  (\*  $R(i) = S(j)$ , so we skip one of the duplicate tuples \*)  
 }  
 if  $(i \leq n)$  then add tuples  $R(i)$  to  $R(n)$  to  $T$ ;  
 if  $(j \leq m)$  then add tuples  $S(j)$  to  $S(m)$  to  $T$ ;
- (d) sort the tuples in  $R$  and  $S$  using the same unique sort attributes;  
 set  $i \leftarrow 1, j \leftarrow 1$ ;  
 while  $(i \leq n)$  and  $(j \leq m)$   
 do { if  $R(i) > S(j)$   
     then set  $j \leftarrow j + 1$   
     elseif  $R(i) < S(j)$   
     then set  $i \leftarrow i + 1$   
     else { output  $R(j)$  to  $T$ ; (\*  $R(i) = S(j)$ , so we output the tuple \*)  
         set  $i \leftarrow i + 1, j \leftarrow j + 1$   
     }  
 }  
 }
- (e) sort the tuples in  $R$  and  $S$  using the same unique sort attributes;  
 set  $i \leftarrow 1, j \leftarrow 1$ ;  
 while  $(i \leq n)$  and  $(j \leq m)$   
 do { if  $R(i) > S(j)$   
     then set  $j \leftarrow j + 1$   
     elseif  $R(i) < S(j)$   
     then { output  $R(i)$  to  $T$ ; (\*  $R(i)$  has no matching  $S(j)$ , so output  $R(i)$  \*)  
         set  $i \leftarrow i + 1$   
     }  
     else set  $i \leftarrow i + 1, j \leftarrow j + 1$   
 }  
 }  
 if  $(i \leq n)$  then add tuples  $R(i)$  to  $R(n)$  to  $T$ ;

### 18.4.2 How Buffer Space and Choice of Outer-Loop File Affect Performance of Nested-Loop Join

The buffer space available has an important effect on some of the join algorithms. First, let us consider the nested-loop approach (J1). Looking again at the operation OP6 above, assume that the number of buffers available in main memory for implementing the join is  $n_B = 7$  blocks (buffers). Recall that we assume that each memory buffer is the same size as one disk block. For illustration, assume that the DEPARTMENT file consists of  $r_D = 50$  records stored in  $b_D = 10$  disk blocks and that the EMPLOYEE file consists of  $r_E = 6,000$  records stored in  $b_E = 2,000$  disk blocks. It is advantageous to read as many blocks as possible at a time into memory from the file whose records are used for the outer loop. Note that keeping one block for reading from the inner file and one block for writing to the output file,  $n_B - 2$  blocks are available to read from the outer relation. The algorithm can then read one block at a time for the inner-loop file and use its records to **probe** (that is, search) the outer-loop blocks that are currently in main memory for matching records. This reduces the total number of block accesses. An extra buffer in main memory is needed to contain the resulting records after they are joined, and the contents of this result buffer can be appended to the **result file**—the disk file that will contain the join result—whenever it is filled. This result buffer block then is reused to hold additional join result records.

In the nested-loop join, it makes a difference which file is chosen for the outer loop and which for the inner loop. If EMPLOYEE is used for the outer loop, each block of EMPLOYEE is read once, and the entire DEPARTMENT file (each of its blocks) is read once for *each time* we read in  $(n_B - 2)$  blocks of the EMPLOYEE file. We get the following formulas for the number of disk blocks that are read from disk to main memory:

Total number of blocks accessed (read) for outer-loop file =  $b_E$

Number of times  $(n_B - 2)$  blocks of outer file are loaded into main memory =  $\lceil b_E / (n_B - 2) \rceil$

Total number of blocks accessed (read) for inner-loop file =  $b_D * \lceil b_E / (n_B - 2) \rceil$

Hence, we get the following total number of block read accesses:

$$b_E + (\lceil b_E / (n_B - 2) \rceil * b_D) = 2000 + (\lceil (2000/5) \rceil * 10) = 6000 \text{ block accesses}$$

On the other hand, if we use the DEPARTMENT records in the outer loop, by symmetry we get the following total number of block accesses:

$$b_D + (\lceil b_D / (n_B - 2) \rceil * b_E) = 10 + (\lceil (10/5) \rceil * 2000) = 4010 \text{ block accesses}$$

The join algorithm uses a buffer to hold the joined records of the result file. Once the buffer is filled, it is written to disk and its contents are appended to the result file, and then refilled with join result records.<sup>12</sup>

<sup>12</sup>If we reserve two buffers for the result file, double buffering can be used to speed the algorithm (see Section 16.3).

If the result file of the join operation has  $b_{RES}$  disk blocks, each block is written once to disk, so an additional  $b_{RES}$  block accesses (writes) should be added to the preceding formulas in order to estimate the total cost of the join operation. The same holds for the formulas developed later for other join algorithms. As this example shows, it is advantageous to use the file *with fewer blocks* as the outer-loop file in the nested-loop join.

### 18.4.3 How the Join Selection Factor Affects Join Performance

Another factor that affects the performance of a join, particularly the single-loop method J2, is the fraction of records in one file that will be joined with records in the other file. We call this the **join selection factor**<sup>13</sup> of a file with respect to an equijoin condition with another file. This factor depends on the particular equijoin condition between the two files. To illustrate this, consider the operation OP7, which joins each DEPARTMENT record with the EMPLOYEE record for the manager of that department. Here, each DEPARTMENT record (there are 50 such records in our example) will be joined with a *single* EMPLOYEE record, but many EMPLOYEE records (the 5,950 of them that do not manage a department) will not be joined with any record from DEPARTMENT.

Suppose that secondary indexes exist on both the attributes Ssn of EMPLOYEE and Mgr\_ssn of DEPARTMENT, with the number of index levels  $x_{Ssn} = 4$  and  $x_{Mgr\_ssn} = 2$ , respectively. We have two options for implementing method J2. The first retrieves each EMPLOYEE record and then uses the index on Mgr\_ssn of DEPARTMENT to find a matching DEPARTMENT record. In this case, no matching record will be found for employees who do not manage a department. The number of block accesses for this case is approximately:

$$b_E + (r_E * (x_{Mgr\_ssn} + 1)) = 2000 + (6000 * 3) = 20,000 \text{ block accesses}$$

The second option retrieves each DEPARTMENT record and then uses the index on Ssn of EMPLOYEE to find a matching manager EMPLOYEE record. In this case, every DEPARTMENT record will have one matching EMPLOYEE record. The number of block accesses for this case is approximately:

$$b_D + (r_D * (x_{Ssn} + 1)) = 10 + (50 * 5) = 260 \text{ block accesses}$$

The second option is more efficient because the join selection factor of DEPARTMENT *with respect to the join condition*  $Ssn = Mgr\_ssn$  is 1 (every record in DEPARTMENT will be joined), whereas the join selection factor of EMPLOYEE with respect to the same join condition is  $(50/6,000)$ , or 0.008 (only 0.8% of the records in EMPLOYEE will be joined). For method J2, either the smaller file or the file that has a match for every record (that is, the file with the high join selection factor) should be used in the (single) join loop. It is also possible to create an index specifically for performing the join operation if one does not already exist.

<sup>13</sup>This is different from the *join selectivity*, which we will discuss in Chapter 19.

The sort-merge join J3 is quite efficient if both files are already sorted by their join attribute. Only a single pass is made through each file. Hence, the number of blocks accessed is equal to the sum of the numbers of blocks in both files. For this method, both OP6 and OP7 would need  $b_E + b_D = 2,000 + 10 = 2,010$  block accesses. However, both files are required to be ordered by the join attributes; if one or both are not, a sorted copy of each file must be created specifically for performing the join operation. If we roughly estimate the cost of sorting an external file by  $(b \log_2 b)$  block accesses, and if both files need to be sorted, the total cost of a sort-merge join can be estimated by  $(b_E + b_D + b_E \log_2 b_E + b_D \log_2 b_D)$ .<sup>14</sup>

#### 18.4.4 General Case for Partition-Hash Join

The hash-join method J4 is also efficient. In this case, only a single pass is made through each file, whether or not the files are ordered. If the hash table for the smaller of the two files can be kept entirely in main memory after hashing (partitioning) on its join attribute, the implementation is straightforward. If, however, the partitions of both files must be stored on disk, the method becomes more complex, and a number of variations to improve the efficiency have been proposed. We discuss two techniques: the general case of *partition-hash join* and a variation called *hybrid hash-join algorithm*, which has been shown to be efficient.

In the general case of **partition-hash join**, each file is first partitioned into  $M$  partitions using the same **partitioning hash function** on the join attributes. Then, each pair of corresponding partitions is joined. For example, suppose we are joining relations  $R$  and  $S$  on the join attributes  $R.A$  and  $S.B$ :

$$R \bowtie_{A=B} S$$

In the **partitioning phase**,  $R$  is partitioned into the  $M$  partitions  $R_1, R_2, \dots, R_M$ , and  $S$  into the  $M$  partitions  $S_1, S_2, \dots, S_M$ . The property of each pair of corresponding partitions  $R_i, S_i$  with respect to the join operation is that records in  $R_i$  *only need to be joined* with records in  $S_i$ , and vice versa. This property is ensured by using the *same hash function* to partition both files on their join attributes—attribute  $A$  for  $R$  and attribute  $B$  for  $S$ . The minimum number of in-memory buffers needed for the **partitioning phase** is  $M + 1$ . Each of the files  $R$  and  $S$  is partitioned separately. During partitioning of a file,  $M$  in-memory buffers are allocated to store the records that hash to each partition, and one additional buffer is needed to hold one block at a time of the input file being partitioned. Whenever the in-memory buffer for a partition gets filled, its contents are appended to a **disk subfile** that stores the partition. The partitioning phase has *two iterations*. After the first iteration, the first file  $R$  is partitioned into the subfiles  $R_1, R_2, \dots, R_M$ , where all the records that hashed to the same buffer are in the same partition. After the second iteration, the second file  $S$  is similarly partitioned.

In the second phase, called the **joining** or **probing phase**,  $M$  iterations are needed. During iteration  $i$ , two corresponding partitions  $R_i$  and  $S_i$  are joined. The minimum

<sup>14</sup>We can use the more accurate formulas from Section 19.5 if we know the number of available buffers for sorting.

number of buffers needed for iteration  $i$  is the number of blocks in the smaller of the two partitions, say  $R_i$ , plus two additional buffers. If we use a nested-loop join during iteration  $i$ , the records from the smaller of the two partitions  $R_i$  are copied into memory buffers; then all blocks from the other partition  $S_i$  are read—one at a time—and each record is used to **probe** (that is, search) partition  $R_i$  for matching record(s). Any matching records are joined and written into the result file. To improve the efficiency of in-memory probing, it is common to use an *in-memory hash table* for storing the records in partition  $R_i$  by using a *different* hash function from the partitioning hash function.<sup>15</sup>

We can approximate the cost of this partition hash-join as  $3 * (b_R + b_S) + b_{RES}$  for our example, since each record is read once and written back to disk once during the partitioning phase. During the joining (probing) phase, each record is read a second time to perform the join. The *main difficulty* of this algorithm is to ensure that the partitioning hash function is **uniform**—that is, the partition sizes are nearly equal in size. If the partitioning function is **skewed** (nonuniform), then some partitions may be too large to fit in the available memory space for the second joining phase.

Notice that if the available in-memory buffer space  $n_B > (b_R + 2)$ , where  $b_R$  is the number of blocks for the *smaller* of the two files being joined, say  $R$ , then there is no reason to do partitioning since in this case the join can be performed entirely in memory using some variation of the nested-loop join based on hashing and probing. For illustration, assume we are performing the join operation OP6, repeated below:

OP6: EMPLOYEE  $\bowtie_{Dno=Dnumber}$  DEPARTMENT

In this example, the smaller file is the DEPARTMENT file; hence, if the number of available memory buffers  $n_B > (b_D + 2)$ , the whole DEPARTMENT file can be read into main memory and organized into a hash table on the join attribute. Each EMPLOYEE block is then read into a buffer, and each EMPLOYEE record in the buffer is hashed on its join attribute and is used to *probe* the corresponding in-memory bucket in the DEPARTMENT hash table. If a matching record is found, the records are joined, and the result record(s) are written to the result buffer and eventually to the result file on disk. The cost in terms of block accesses is hence  $(b_D + b_E)$ , plus  $b_{RES}$ —the cost of writing the result file.

### 18.4.5 Hybrid Hash-Join

The **hybrid hash-join algorithm** is a variation of partition hash-join, where the *joining* phase for *one of the partitions* is included in the *partitioning* phase. To illustrate this, let us assume that the size of a memory buffer is one disk block; that  $n_B$  such buffers are *available*; and that the partitioning hash function used is  $h(K) = K \bmod M$ , so that  $M$  partitions are being created, where  $M < n_B$ . For illustration, assume we are performing the join operation OP6. In the *first pass* of the partitioning phase, when the hybrid hash-join algorithm is partitioning the smaller of the two files

<sup>15</sup>If the hash function used for partitioning is used again, all records in a partition will hash to the same bucket again.

(DEPARTMENT in OP6), the algorithm divides the buffer space among the  $M$  partitions such that all the blocks of the *first partition* of DEPARTMENT completely reside in main memory. For each of the other partitions, only a single in-memory buffer—whose size is one disk block—is allocated; the remainder of the partition is written to disk as in the regular partition-hash join. Hence, at the end of the *first pass of the partitioning phase*, the first partition of DEPARTMENT resides wholly in main memory, whereas each of the other partitions of DEPARTMENT resides in a disk subfile.

For the second pass of the partitioning phase, the records of the second file being joined—the larger file, EMPLOYEE in OP6—are being partitioned. If a record hashes to the *first partition*, it is joined with the matching record in DEPARTMENT and the joined records are written to the result buffer (and eventually to disk). If an EMPLOYEE record hashes to a partition other than the first, it is partitioned normally and stored to disk. Hence, at the end of the second pass of the partitioning phase, all records that hash to the first partition have been joined. At this point, there are  $M - 1$  pairs of partitions on disk. Therefore, during the second **joining** or **probing** phase,  $M - 1$  iterations are needed instead of  $M$ . The goal is to join as many records during the partitioning phase so as to save the cost of storing those records on disk and then rereading them a second time during the joining phase.

## 18.5 Algorithms for PROJECT and Set Operations

A PROJECT operation  $\pi_{\langle \text{attribute list} \rangle}(R)$  from relational algebra implies that after projecting  $R$  on only the columns in the list of attributes, any duplicates are removed by treating the result strictly as a set of tuples. However, the SQL query:

```
SELECT Salary
FROM EMPLOYEE
```

produces a list of salaries of all employees. If there are 10,000 employees and only 80 distinct values for salary, it produces a one column result with 10,000 tuples. This operation is done by simple linear search by making a complete pass through the table.

Getting the true effect of the relational algebra  $\pi_{\langle \text{attribute list} \rangle}(R)$  operator is straightforward to implement if  $\langle \text{attribute list} \rangle$  includes a key of relation  $R$ , because in this case the result of the operation will have the same number of tuples as  $R$ , but with only the values for the attributes in  $\langle \text{attribute list} \rangle$  in each tuple. If  $\langle \text{attribute list} \rangle$  does not include a key of  $R$ , *duplicate tuples must be eliminated*. This can be done by sorting the result of the operation and then eliminating duplicate tuples, which appear consecutively after sorting. A sketch of the algorithm is given in Figure 18.3(b). Hashing can also be used to eliminate duplicates: as each record is hashed and inserted into a bucket of the hash file in memory, it is checked against those records already in the bucket; if it is a duplicate, it is not inserted in the bucket. It is useful to recall here that in SQL queries, the default is not to eliminate duplicates from the query result; duplicates are eliminated from the query result only if the keyword DISTINCT is included.

Set operations—UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT—are sometimes expensive to implement, since UNION, INTERSECTION, MINUS or SET DIFFERENCE are set operators and must always return distinct results.

In particular, the CARTESIAN PRODUCT operation  $R \times S$  is expensive because its result includes a record for each combination of records from  $R$  and  $S$ . Also, each record in the result includes all attributes of  $R$  and  $S$ . If  $R$  has  $n$  records and  $j$  attributes, and  $S$  has  $m$  records and  $k$  attributes, the result relation for  $R \times S$  will have  $n * m$  records and each record will have  $j + k$  attributes. Hence, it is important to avoid the CARTESIAN PRODUCT operation and to substitute other operations such as join during query optimization. The other three set operations—UNION, INTERSECTION, and SET DIFFERENCE<sup>16</sup>—apply only to **type-compatible** (or union-compatible) relations, which have the same number of attributes and the same attribute domains. The customary way to implement these operations is to use variations of the **sort-merge technique**: the two relations are sorted on the same attributes, and, after sorting, a single scan through each relation is sufficient to produce the result. For example, we can implement the UNION operation,  $R \cup S$ , by scanning and merging both sorted files concurrently, and whenever the same tuple exists in both relations, only one is kept in the merged result. For the INTERSECTION operation,  $R \cap S$ , we keep in the merged result only those tuples that appear in *both sorted relations*. Figure 18.3(c) to (e) sketches the implementation of these operations by sorting and merging. Some of the details are not included in these algorithms.

**Hashing** can also be used to implement UNION, INTERSECTION, and SET DIFFERENCE. One table is first scanned and then partitioned into an in-memory hash table with buckets, and the records in the other table are then scanned one at a time and used to probe the appropriate partition. For example, to implement  $R \cup S$ , first hash (partition) the records of  $R$ ; then, hash (probe) the records of  $S$ , but do not insert duplicate records in the buckets. To implement  $R \cap S$ , first partition the records of  $R$  to the hash file. Then, while hashing each record of  $S$ , probe to check if an identical record from  $R$  is found in the bucket, and if so add the record to the result file. To implement  $R - S$ , first hash the records of  $R$  to the hash file buckets. While hashing (probing) each record of  $S$ , if an identical record is found in the bucket, remove that record from the bucket.

### 18.5.1 Use of Anti-Join for SET DIFFERENCE (or EXCEPT or MINUS in SQL)

The MINUS operator in SQL is transformed into an anti-join (which we introduced in Section 18.1) as follows. Suppose we want to find out which departments have no employees in the schema of Figure 5.5:

```
Select Dnumber from DEPARTMENT MINUS Select Dno from EMPLOYEE;
```

<sup>16</sup>SET DIFFERENCE is called MINUS or EXCEPT in SQL.



can be converted into the following:

```
SELECT DISTINCT DEPARTMENT.Dnumber
FROM DEPARTMENT, EMPLOYEE
WHERE DEPARTMENT.Dnumber A = EMPLOYEE.Dno
```

We used the nonstandard notation for anti-join, “A=”, where DEPARTMENT is on the left of anti-join and EMPLOYEE is on the right.

In SQL, there are two variations of these set operations. The operations UNION, INTERSECTION, and EXCEPT or MINUS (the SQL keywords for the SET DIFFERENCE operation) apply to traditional sets, where no duplicate records exist in the result. The operations UNION ALL, INTERSECTION ALL, and EXCEPT ALL apply to multisets (or bags). Thus, going back to the database of Figure 5.5, consider a query that finds all departments that employees are working on where at least one project exists controlled by that department, and this result is written as:

```
SELECT Dno from EMPLOYEE
INTERSECT ALL
SELECT Dum from PROJECT
```

This would not eliminate any duplicates of Dno from EMPLOYEE while performing the INTERSECTION. If all 10,000 employees are assigned to departments where some project is present in the PROJECT relation, the result would be the list of all the 10,000 department numbers including duplicates.. This can be accomplished by the semi-join operation we introduced in Section 18.1 as follows:

```
SELECT DISTINCT EMPLOYEE.Dno
FROM DEPARTMENT, EMPLOYEE
WHERE EMPLOYEE.Dno S = DEPARTMENT.Dnumber
```

If INTERSECTION is used without the ALL, then an additional step of duplicate elimination will be required for the selected department numbers.

## 18.6 Implementing Aggregate Operations and Different Types of JOINS

### 18.6.1 Implementing Aggregate Operations

The aggregate operators (MIN, MAX, COUNT, AVERAGE, SUM), when applied to an entire table, can be computed by a table scan or by using an appropriate index, if available. For example, consider the following SQL query:

```
SELECT MAX(Salary)
FROM EMPLOYEE;
```

If an (ascending) B<sup>+</sup>-tree index on Salary exists for the EMPLOYEE relation, then the optimizer can decide on using the Salary index to search for the largest Salary value in the index by following the *rightmost* pointer in each index node from the

root to the rightmost leaf. That node would include the largest Salary value as its *last* entry. In most cases, this would be more efficient than a full table scan of EMPLOYEE, since no actual records need to be retrieved. The MIN function can be handled in a similar manner, except that the *leftmost* pointer in the index is followed from the root to leftmost leaf. That node would include the smallest Salary value as its *first* entry.

The index could also be used for the AVERAGE and SUM aggregate functions, but only if it is a **dense index**—that is, if there is an index entry for every record in the main file. In this case, the associated computation would be applied to the values in the index. For a **nondense index**, the actual number of records associated with each index value must be used for a correct computation. This can be done if the *number of records associated with each value* in the index is stored in each index entry. For the COUNT aggregate function, the number of values can be also computed from the index in a similar manner. If a COUNT(\*) function is applied to a whole relation, the number of records currently in each relation are typically stored in the catalog, and so the result can be retrieved directly from the catalog.

When a GROUP BY clause is used in a query, the aggregate operator must be applied separately to each group of tuples as partitioned by the grouping attribute. Hence, the table must first be partitioned into subsets of tuples, where each partition (group) has the same value for the grouping attributes. In this case, the computation is more complex. Consider the following query:

```
SELECT      Dno, AVG(Salary)
FROM        EMPLOYEE
GROUP BY    Dno;
```

The usual technique for such queries is to first use either **sorting** or **hashing** on the grouping attributes to partition the file into the appropriate groups. Then the algorithm computes the aggregate function for the tuples in each group, which have the same grouping attribute(s) value. In the sample query, the set of EMPLOYEE tuples for each department number would be grouped together in a partition and the average salary computed for each group.

Notice that if a **clustering index** (see Chapter 17) exists on the grouping attribute(s), then the records are *already partitioned* (grouped) into the appropriate subsets. In this case, it is only necessary to apply the computation to each group.

## 18.6.2 Implementing Different Types of JOINS

In addition to the standard JOIN (also called INNER JOIN in SQL), there are variations of JOIN that are frequently used. Let us briefly consider three of them below: outer joins, semi-joins, and anti-joins.

**Outer Joins.** In Section 6.4, we discussed the *outer join operation*, with its three variations: left outer join, right outer join, and full outer join. In Chapter 5, we

discussed how these operations can be specified in SQL. The following is an example of a left outer join operation in SQL:

```
SELECT E.Lname, E.Fname, D.Dname
FROM (EMPLOYEE E LEFT OUTER JOIN DEPARTMENT D ON E.Dno = D.Dnumber);
```

The result of this query is a table of employee names and their associated departments. The table contains the same results as a regular (inner) join, with the exception that if an EMPLOYEE tuple (a tuple in the *left* relation) *does not have an associated department*, the employee's name will still appear in the resulting table, but the department name would be NULL for such tuples in the query result. Outer join can be looked upon as a combination of inner join and anti-join.

Outer join can be computed by modifying one of the join algorithms, such as nested-loop join or single-loop join. For example, to compute a *left* outer join, we use the left relation as the outer loop or index-based nested loop because every tuple in the left relation must appear in the result. If there are matching tuples in the other relation, the joined tuples are produced and saved in the result. However, if no matching tuple is found, the tuple is still included in the result but is padded with NULL value(s). The sort-merge and hash-join algorithms can also be extended to compute outer joins.

Theoretically, outer join can also be computed by executing a combination of relational algebra operators. For example, the left outer join operation shown above is equivalent to the following sequence of relational operations:

1. Compute the (inner) JOIN of the EMPLOYEE and DEPARTMENT tables.

$$\text{TEMP1} \leftarrow \pi_{\text{Lname, Fname, Dname}} (\text{EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT})$$

2. Find the EMPLOYEE tuples that do not appear in the (inner) JOIN result.

$$\text{TEMP2} \leftarrow \pi_{\text{Lname, Fname}} (\text{EMPLOYEE}) - \pi_{\text{Lname, Fname}} (\text{TEMP1})$$

This minus operation can be achieved by performing an anti-join on Lname, Fname between EMPLOYEE and TEMP1, as we discussed above in Section 18.5.2.

3. Pad each tuple in TEMP2 with a NULL Dname field.

$$\text{TEMP2} \leftarrow \text{TEMP2} \times \text{NULL}$$

4. Apply the UNION operation to TEMP1, TEMP2 to produce the LEFT OUTER JOIN result.

$$\text{RESULT} \leftarrow \text{TEMP1} \cup \text{TEMP2}$$

The cost of the outer join as computed above would be the sum of the costs of the associated steps (inner join, projections, set difference, and union). However, note that step 3 can be done as the temporary relation is being constructed in step 2; that is, we can simply pad each resulting tuple with a NULL. In addition, in step 4, we know that the two operands of the union are disjoint (no common tuples), so there is no need for duplicate elimination. So the preferred method is to use a combination of inner join and anti-join rather than the above steps since the algebraic

approach of projection followed by set difference causes temporary tables to be stored and processed multiple times.

The right outer join can be converted to a left outer join by switching the operands and hence needs no separate discussion. **Full outer join** requires computing the result of inner join and then padding to the result extra tuples arising from unmatched tuples from both the left and right operand relations. Typically, full outer join would be computed by extending sort-merge or hashed join algorithms to account for the unmatched tuples.

**Implementing Semi-Join and Anti-Join.** In Section 18.1, we introduced these types of joins as possible operations to which some queries with nested subqueries get mapped. The purpose is to be able to perform some variant of join instead of evaluating the subquery multiple times. Use of inner join would be invalid in these cases, since for every tuple of the outer relation, the inner join looks for all possible matches on the inner relation. In semi-join, the search stops as soon as the first match is found and the tuple from outer relation is selected; in anti-join, search stops as soon as the first match is found and the tuple from outer relation is rejected. Both these types of joins can be implemented as an extension of the join algorithms we discussed in Section 18.4.

**Implementing Non-Equi-Join** Join operation may also be performed when the join condition is one of inequality. In Chapter 6, we referred to this operation as theta-join. This functionality is based on a condition involving any operators, such as  $<$ ,  $>$ ,  $\geq$ ,  $\leq$ ,  $\neq$ , and so on. All of the join methods discussed are again applicable here with the exception that hash-based algorithms cannot be used.

## 18.7 Combining Operations Using Pipelining

A query specified in SQL will typically be translated into a relational algebra expression that is *a sequence of relational operations*. If we execute a single operation at a time, we must generate temporary files on disk to hold the results of these temporary operations, creating excessive overhead. Evaluating a query by creating and storing each temporary result and then passing it as an argument for the next operator is called **materialized evaluation**. Each temporary materialized result is then written to disk and adds to the overall cost of query processing.

Generating and storing large temporary files on disk is time-consuming and can be unnecessary in many cases, since these files will immediately be used as input to the next operation. To reduce the number of temporary files, it is common to generate query execution code that corresponds to algorithms for combinations of operations in a query.

For example, rather than being implemented separately, a JOIN can be combined with two SELECT operations on the input files and a final PROJECT operation on the resulting file; all this is implemented by one algorithm with two input files and a single output file. Rather than creating four temporary files, we apply the algorithm directly and get just one result file.

In Section 19.1, we discuss how heuristic relational algebra optimization can group operations together for execution. Combining several operations into one and avoiding the writing of temporary results to disk is called **pipelining** or **stream-based processing**.

It is common to create the query execution code dynamically to implement multiple operations. The generated code for producing the query combines several algorithms that correspond to individual operations. As the result tuples from one operation are produced, they are provided as input for subsequent operations. For example, if a join operation follows two select operations on base relations, the tuples resulting from each select are provided as input for the join algorithm in a **stream** or **pipeline** as they are produced. The corresponding evaluation is considered a **pipelined evaluation**. It has two distinct benefits:

- Avoiding the additional cost and time delay incurred for writing the intermediate results to disk.
- Being able to start generating results as quickly as possible when the root operator is combined with some of the operators discussed in the following section means that the pipelined evaluation can start generating tuples of the result while rest of the pipelined intermediate tables are undergoing processing.

### 18.7.1 Iterators for implementing Physical Operations

Various algorithms for algebraic operations involve reading some input in the form of one or more files, processing it, and generating an output file as a relation. If the operation is implemented in such a way that it outputs one tuple at a time, then it can be regarded as an **iterator**. For example, we can devise a tuple-based implementation of the nested-loop join that will generate a tuple at a time as output. Iterators work in contrast with the materialization approach wherein entire relations are produced as temporary results and stored on disk or main memory and are read back again by the next algorithm. The query plan that contains the query tree may be executed by invoking the iterators in a certain order. Many iterators may be active at one time, thereby passing results up the execution tree and avoiding the need for additional storage of temporary results. The iterator interface typically consists of the following methods:

1. **Open ()**: This method initializes the operator by allocating buffers for its input and output and initializing any data structures needed for the operator. It is also used to pass arguments such as selection conditions needed to perform the operation. It in turn calls **Open()** to get the arguments it needs.
2. **Get\_Next ()**: This method calls the **Get\_next()** on each of its input arguments and calls the code specific to the operation being performed on the inputs. The next output tuple generated is returned and the state of the iterator is updated to keep track of the amount of input processed. When no more tuples can be returned, it places some special value in the output buffer.

3. Close(): This method ends the iteration after all tuples that can be generated have been generated, or the required/demanded number of tuples have been returned. It also calls Close() on the arguments of the iterator.

Each iterator may be regarded as a class for its implementation with the above three methods applicable to each instance of that class. If the operator to be implemented allows a tuple to be completely processed when it is received, it may be possible to use the pipelining strategy effectively. However, if the input tuples need to be examined over multiple passes, then the input has to be received as a materialized relation. This becomes tantamount to the Open () method doing most of the work and the benefit of pipelining not being fully achieved. Some physical operators may not lend themselves to the iterator interface concept and hence may not support pipelining.

The iterator concept may also be applied to access methods. Accessing a B<sup>+</sup>-tree or a hash-based index may be regarded as a function that can be implemented as an iterator; it produces as output a series of tuples that meet the selection condition passed to the Open() method.

## 18.8 Parallel Algorithms for Query Processing

In Chapter 2, we mentioned several variations of the client/server architectures, including two-tier and three-tier architectures. There is another type of architecture, called **parallel database architecture**, that is prevalent for data-intensive applications. We will discuss it in further detail in Chapter 23 in conjunction with distributed databases and the big data and NOSQL emerging technologies.

Three main approaches have been proposed for parallel databases. They correspond to three different hardware configurations of processors and secondary storage devices (disks) to support parallelism. In **shared-memory architecture**, multiple processors are attached to an interconnection network and can access a common main memory region. Each processor has access to the entire memory address space from all machines. The memory access to local memory and local cache is faster; memory access to the common memory is slower. This architecture suffers from interference because as more processors are added, there is increasing contention for the common memory. The second type of architecture is known as **shared-disk architecture**. In this architecture, every processor has its own memory, which is not accessible from other processors. However, every machine has access to all disks through the interconnection network. Every processor may not necessarily have a disk of its own. We discussed two forms of enterprise-level secondary storage systems in Section 16.11. Both storage area networks (SANs) and network attached storage (NAS) fall into the shared-disk architecture and lend themselves to parallel processing. They have different units of data transfer; SANs transfer data in units of blocks or pages to and from disks to processors; NAS behaves like a file server that transfers files using some file transfer protocol. In these systems, as more processors are added, there is more contention for the limited network bandwidth.

The above difficulties have led to **shared-nothing architecture** becoming the most commonly used architecture in parallel database systems. In this architecture, each processor accesses its own main memory and disk storage. When a processor A requests data located on the disk  $D_B$  attached to processor B, processor A sends the request as a message over a network to processor B, which accesses its own disk  $D_B$  and ships the data over the network in a message to processor A. Parallel databases using shared-nothing architecture are relatively inexpensive to build. Today, commodity processors are being connected in this fashion on a rack, and several racks can be connected by an external network. Each processor has its own memory and disk storage.

The shared-nothing architecture affords the possibility of achieving parallelism in query processing at three levels, which we will discuss below: individual operator parallelism, intraquery parallelism, and interquery parallelism. Studies have shown that by allocating more processors and disks, **linear speed-up**—a linear reduction in the time taken for operations—is possible. **Linear scale-up**, on the other hand, refers to being able to give a constant sustained performance by increasing the number of processors and disks proportional to the size of data. Both of these are implicit goals of parallel processing.

### 18.8.1 Operator-Level Parallelism

In the operations that can be implemented with parallel algorithms, one of the main strategies is to partition data across disks. **Horizontal partitioning** of a relation corresponds to distributing the tuples across disks based on some partitioning method. Given  $n$  disks, assigning the  $i$ th tuple to disk  $i \bmod n$  is called **round-robin partitioning**. Under **range partitioning**, tuples are equally distributed (as much as possible) by dividing the range of values of some attribute. For example, employee tuples from the EMPLOYEE relation may be assigned to 10 disks by dividing the age range into 10 ranges—say 22–25, 26–28, 29–30, and so on—such that each has roughly one-tenth of the total number of employees. Range partitioning is a challenging operation and requires a good understanding of the distribution of data along the attribute involved in the range clause. The ranges used for partitioning are represented by the **range vector**. With **hash partitioning**, tuple  $i$  is assigned to the disk  $h(i)$ , where  $h$  is the hashing function. Next, we briefly discuss how parallel algorithms are designed for various individual operations.

**Sorting.** If the data has been range partitioned on an attribute—say, age—into  $n$  disks on  $n$  processors, then to sort the entire relation on age, each partition can be sorted separately in parallel and the results can be concatenated. This potentially causes close to an  $n$ -fold reduction in the overall sorting time. If the relation has been partitioned using another scheme, the following approaches are possible:

- Repartition the relation by using range partitioning on the same attribute that is the target for sorting; then sort each partition individually followed by concatenation, as mentioned above.
- Use a parallel version of the external sort-merge algorithm shown in Figure 18.2.



**Selection.** For a selection based on some condition, if the condition is an equality condition,  $\langle A = v \rangle$  and the same attribute  $A$  has been used for range partitioning, the selection can be performed on only that partition to which the value  $v$  belongs. In other cases, the selection would be performed in parallel on all the processors and the results merged. If the selection condition is  $v1 \leq A \leq v2$  and attribute  $A$  is used for range partitioning, then the range of values  $(v1, v2)$  must overlap a certain number of partitions. The selection operation needs to be performed only in those processors in parallel.

**Projection and Duplicate Elimination.** Projection without duplicate elimination can be achieved by performing the operation in parallel as data is read from each partition. Duplicate elimination can be achieved by sorting the tuples and discarding duplicates. For sorting, any of the techniques mentioned above can be used based on how the data is partitioned.

**Join.** The basic idea of parallel join is to split the relations to be joined, say  $R$  and  $S$ , in such a way that the join is divided into multiple  $n$  smaller joins, and then perform these smaller joins in parallel on  $n$  processors and take a union of the result. Next, we discuss the various techniques involved to achieve this.

- a. **Equality-based partitioned join:** If both the relations  $R$  and  $S$  are partitioned into  $n$  partitions on  $n$  processors such that partition  $r_i$  and partition  $s_i$  are both assigned to the same processor  $P_i$ , then the join can be computed locally provided the join is an equality join or natural join. Note that the partitions must be non-overlapping on the join key; in that sense, the partitioning is a strict set-theoretic partitioning. Furthermore, the attribute used in the join condition must also satisfy these conditions:
  - It is the same as that used for range partitioning, and the ranges used for each partition are also the same for both  $R$  and  $S$ . Or,
  - It is the same as that used to partition into  $n$  partitions using hash partitioning. The same hash function must be used for  $R$  and  $S$ . If the distributions of values of the joining attribute are different in  $R$  and  $S$ , it is difficult to come up with a range vector that will uniformly distribute both  $R$  and  $S$  into equal partitions. Ideally, the size of  $|r_i| + |s_i|$  should be even for all partitions  $i$ . Otherwise, if there is too much data skew, then the benefits of parallel processing are not fully achieved. The local join at each processor may be performed using any of the techniques discussed for join: sort merge, nested loop, and hash join.
- b. **Inequality join with partitioning and replication:** If the join condition is an inequality condition, involving  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $\neq$ , and so on, then it is not possible to partition  $R$  and  $S$  in such a way that the  $i$ th partition of  $R$ —namely,  $r_i$ —joins the  $j$ th partition of  $S$ —namely,  $s_j$  only. Such a join can be parallelized in two ways:
  - *Asymmetric case:* Partitioning a relation  $R$  using one of the partitioning schemes; replicating one of the relations (say  $S$ ) to all the  $n$  partitions; and performing the join between  $r_i$  and the entire  $S$  at processor  $P_i$ . This method is preferred when  $S$  is much smaller than  $R$ .



- *Symmetric case:* Under this general method, which is applicable to any type of join, both  $R$  and  $S$  are partitioned.  $R$  is partitioned  $n$  ways, and  $S$  is partitioned  $m$  ways. A total of  $m * n$  processors are used for the parallel join. These partitions are appropriately replicated so that processors  $P_{0,0}$  thru  $P_{n-1,m-1}$  (total of  $m * n$  processors) can perform the join locally. The processor  $P_{i,j}$  performs the join of  $r_i$  with  $s_j$  using any of the join techniques. The system replicates the partition  $r_i$  to processors  $P_{i,0}, P_{i,1}$  thru  $P_{i,m-1}$ . Similarly, partition  $s_j$  is replicated to processors  $P_{0,j}, P_{1,j}, P_{n-1,j}$ . In general, partitioning with replication has a higher cost than just partitioning; thus partitioning with replication costs more in the case of an equijoin.
- c. **Parallel partitioned hash join:** The partitioned hash join we described as algorithm J4 in Section 18.4 can be parallelized. The idea is that when  $R$  and  $S$  are large relations, even if we partition each relation into  $n$  partitions equaling the number of processors, the local join at each processor can still be costly. This join proceeds as follows; assume that  $s$  is the smaller of  $r$  and  $s$ :
  1. Using a hash function  $h1$  on the join attribute, map each tuple of relations  $r$  and  $s$  to one of the  $n$  processors. Let  $r_i$  and  $s_i$  be the partitions hashed to  $P_i$ . First, read the  $s$  tuples at each processor on its local disk and map them to the appropriate processor using  $h1$ .
  2. Within each processor  $P_i$ , the tuples of  $S$  received in step 1 are partitioned using a different hash function  $h2$  to, say,  $k$  buckets. This step is identical to the partitioning phase of the partitioned hash algorithm we described as J4 in Section 18.4.
  3. Read the  $r$  tuples from each local disk at each processor and map them to the appropriate processor using hashing function  $h1$ . As they are received at each processor, the processor partitions them using the same hash function  $h2$  used in step 2 for the  $k$  buckets; this process is just as in the probing phase of algorithm J4.
  4. The processor  $P_i$  executes the partitioned hash algorithm locally on the partitions  $r_i$  and  $s_i$  using the joining phase on the  $k$  buckets (as described in algorithm J4) and produces a join result.

The results from all processors  $P_i$  are independently computed and unioned to produce the final result.

**Aggregation.** Aggregate operations with grouping are achieved by partitioning on the grouping attribute and then computing the aggregate function locally at each processor using any of the uni-processor algorithms. Either range partitioning or hash partitioning can be used.

**Set Operations.** For union, intersection, and set difference operations, if the argument relations  $R$  and  $S$  are partitioned using the same hash function, they can be done in parallel on each processor. If the partitioning is based on unmatched criteria,  $R$  and  $S$  may need to be redistributed using an identical hash function.

### 18.8.2 Intraquery Parallelism

We have discussed how each individual operation may be executed by distributing the data among multiple processors and performing the operation in parallel on those processors. A query execution plan can be modeled as a graph of operations. To achieve a parallel execution of a query, one approach is to use a parallel algorithm for each operation involved in the query, with appropriate partitioning of the data input to that operation. Another opportunity to parallelize comes from the evaluation of an operator tree where some of the operations may be executed in parallel because they do not depend on one another. These operations may be executed on separate processors. If the output of one of the operations can be generated tuple-by-tuple and fed into another operator, the result is **pipelined parallelism**. An operator that does not produce any output until it has consumed all its inputs is said to **block the pipelining**.

### 18.8.3 Interquery Parallelism

Interquery parallelism refers to the execution of multiple queries in parallel. In shared-nothing or shared-disk architectures, this is difficult to achieve. Activities of locking, logging, and so on among processors (see the chapters in Part 9 on Transaction Processing) must be coordinated, and simultaneous conflicting updates of the same data by multiple processors must be avoided. There must be **cache coherency**, which guarantees that the processor updating a page has the latest version of that page in the buffer. The cache-coherency and concurrency control protocols (see Chapter 21) must work in coordination as well.

The main goal behind interquery parallelism is to scale up (i.e., to increase the overall rate at which queries or transactions can be processed by increasing the number of processors). Because single-processor multiuser systems themselves are designed to support concurrency control among transactions with the goal of increasing transaction throughput (see Chapter 21), database systems using shared memory parallel architecture can achieve this type of parallelism more easily without significant changes.

From the above discussion it is clear that we can speed up the query execution by performing various operations, such as sorting, selection, projection, join, and aggregate operations, individually using their parallel execution. We may achieve further speed-up by executing parts of the query tree that are independent in parallel on different processors. However, it is difficult to achieve interquery parallelism in shared-nothing parallel architectures. One area where the shared-disk architecture has an edge is that it has a more general applicability, since it, unlike the shared-nothing architecture, does not require data to be stored in a partitioned manner. Current SAN- and NAS-based systems afford this advantage. A number of parameters—such as available number of processors and available buffer space—play a role in determining the overall speed-up. A detailed discussion of the effect of these parameters is outside our scope.

## 18.9 Summary

In this chapter, we gave an overview of the techniques used by DBMSs in processing high-level queries. We first discussed how SQL queries are translated into relational algebra. We introduced the operations of semi-join and anti-join, to which certain nested queries are mapped to avoid doing the regular inner join. We discussed external sorting, which is commonly needed during query processing to order the tuples of a relation while dealing with aggregation, duplicate elimination, and so forth. We considered various cases of selection and discussed the algorithms employed for simple selection based on one attribute and complex selections using conjunctive and disjunctive clauses. Many techniques were discussed for the different selection types, including linear and binary search, use of  $B^+$ -tree index, bitmap indexes, clustering index, and functional index. The idea of selectivity of conditions and the typical information placed in a DBMS catalog was discussed. Then we considered the join operation in detail and proposed algorithms called nested-loop join, index-based nested-loop join, sort-merge join, and hash join.

We gave illustrations of how buffer space, join selection factor, and inner-outer relation choice affect the performance of the join algorithms. We also discussed the hybrid hash algorithm, which avoids some of the cost of writing during the joining phase. We discussed algorithms for projection and set operations as well as algorithms for aggregation. Then we discussed the algorithms for different types of joins, including outer joins, semi-join, anti-join, and non-equi-join. We also discussed how operations can be combined during query processing to create pipelined or stream-based execution instead of materialized execution. We introduced how operators may be implemented using the iterator concept. We ended the discussion of query processing strategies with a quick introduction to the three types of parallel database system architectures. Then we briefly summarized how parallelism can be achieved at the individual operations level and discussed intraquery and interquery parallelism as well.

## Review Questions

- 18.1. Discuss the reasons for converting SQL queries into relational algebra queries before optimization is done.
- 18.2. Discuss semi-join and anti-join as operations to which nested queries may be mapped; provide an example of each.
- 18.3. How are large tables that do not fit in memory sorted? Give the overall procedure.
- 18.4. Discuss the different algorithms for implementing each of the following relational operators and the circumstances under which each algorithm can be used: SELECT, JOIN, PROJECT, UNION, INTERSECT, SET DIFFERENCE, CARTESIAN PRODUCT.
- 18.4. Give examples of a conjunctive selection and a disjunctive selection query and discuss how there may be multiple options for their execution.

- 18.5. Discuss alternative ways of eliminating duplicates when a “SELECT Distinct <attribute>” query is evaluated.
- 18.6. How are aggregate operations implemented?
- 18.7. How are outer join and non-equi-join implemented?
- 18.8. What is the iterator concept? What methods are part of an iterator?
- 18.9. What are the three types of parallel architectures applicable to database systems? Which one is most commonly used?
- 18.10. What are the parallel implementations of join?
- 18.11. What are intraquery and interquery parallelisms? Which one is harder to achieve in the shared-nothing architecture? Why?
- 18.12. Under what conditions is pipelined parallel execution of a sequence of operations prevented?

## Exercises

- 18.13. Consider SQL queries Q1, Q8, Q1B, and Q4 in Chapter 6 and Q27 in Chapter 7.
  - a. Draw at least two query trees that can represent *each* of these queries. Under what circumstances would you use each of your query trees?
  - b. Draw the initial query tree for each of these queries, and then show how the query tree is optimized by the algorithm outlined in Section 18.7.
  - c. For each query, compare your own query trees of part (a) and the initial and final query trees of part (b).
- 18.14. A file of 4,096 blocks is to be sorted with an available buffer space of 64 blocks. How many passes will be needed in the merge phase of the external sort-merge algorithm?
- 18.15. Can a nondense index be used in the implementation of an aggregate operator? Why or why not? Illustrate with an example.
- 18.16. Extend the sort-merge join algorithm to implement the LEFT OUTER JOIN operation.

## Selected Bibliography

We will give references to the literature for the query processing and optimization area together at the end of Chapter 19. Thus the Chapter 19 references apply to this chapter and the next chapter. It is difficult to separate the literature that addresses just query processing strategies and algorithms from the literature that discusses the optimization area.

This page intentionally left blank

## Query Optimization

In this chapter,<sup>1</sup> we will assume that the reader is already familiar with the strategies for query processing in relational DBMSs that we discussed in the previous chapter. The goal of query optimization is to select the best possible strategy for query evaluation. As we said before, the term *optimization* is a misnomer because the chosen execution plan may not always be the most optimal plan possible. The primary goal is to arrive at the most efficient and cost-effective plan using the available information about the schema and the content of relations involved, and to do so in a reasonable amount of time. Thus a proper way to describe **query optimization** would be that it is an activity conducted by a query optimizer in a DBMS to select the best available strategy for executing the query.

This chapter is organized as follows. In Section 19.1 we describe the notation for mapping of the queries from SQL into query trees and graphs. Most RDBMSs use an internal representation of the query as a tree. We present heuristics to transform the query into a more efficient equivalent form followed by a general procedure for applying those heuristics. In Section 19.2, we discuss the conversion of queries into execution plans. We discuss nested subquery optimization. We also present examples of query transformation in two cases: merging of views in Group By queries and transformation of Star Schema queries that arise in data warehouses. We also briefly discuss materialized views. Section 19.3 is devoted to a discussion of selectivity and result-size estimation and presents a cost-based approach to optimization. We revisit the information in the system catalog that we presented in Section 18.3.4 earlier and present histograms. Cost models for selection and join operation are presented in Sections 19.4 and 19.5. We discuss the join ordering problem, which is a critical one, in some detail in Section 19.5.3. Section 19.6 presents an example of cost-based optimization. Section 19.7 discusses some additional issues related to

---

<sup>1</sup>The substantial contribution of Rafi Ahmed to this chapter is appreciated.

query optimization. Section 19.8 is devoted to a discussion of query optimization in data warehouses. Section 19.9 gives an overview of query optimization in Oracle. Section 19.10 briefly discusses semantic query optimization. We end the chapter with a summary in Section 19.11.

## 19.1 Query Trees and Heuristics for Query Optimization

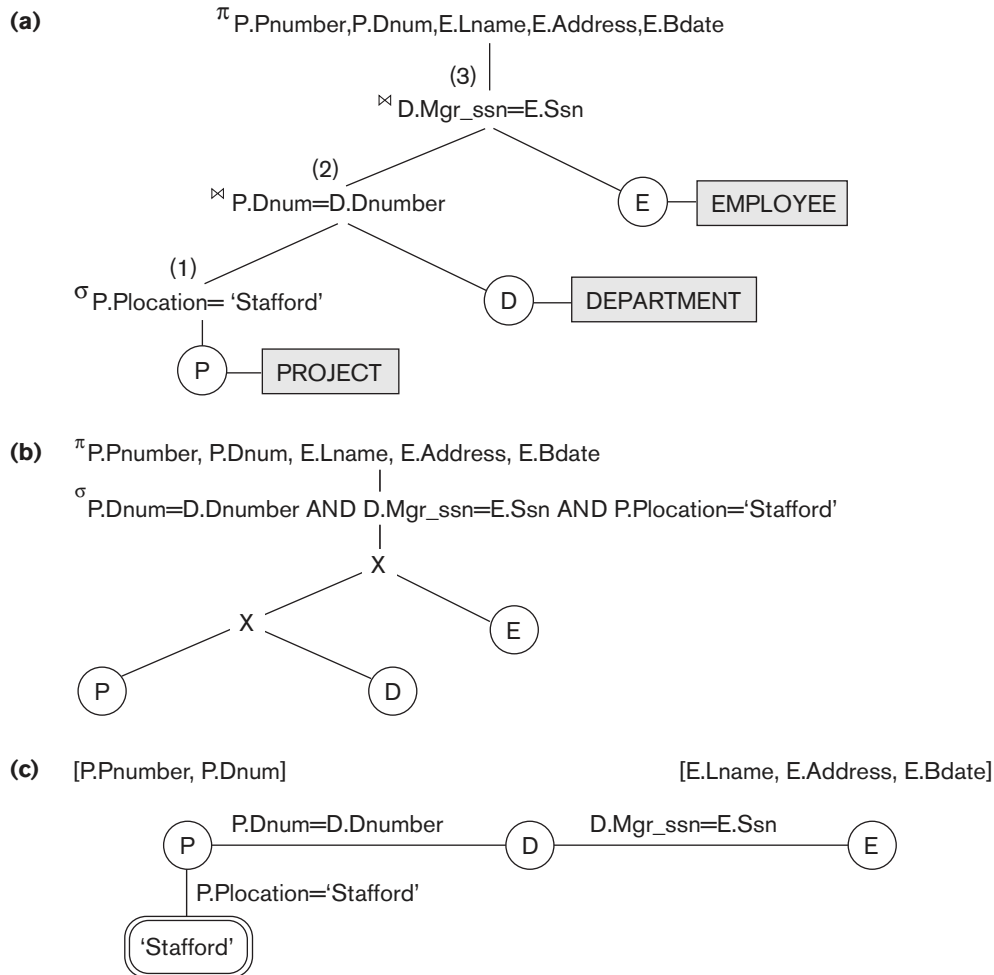
In this section, we discuss optimization techniques that apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data structure—to improve its expected performance. The scanner and parser of an SQL query first generate a data structure that corresponds to an *initial query representation*, which is then optimized according to heuristic rules. This leads to an *optimized query representation*, which corresponds to the query execution strategy. Following that, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

One of the main **heuristic rules** is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations, because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.

In Section 19.1.1, we reiterate the query tree and query graph notations that we introduced earlier in the context of relational algebra and calculus in Sections 8.3.5 and 8.6.5, respectively. These can be used as the basis for the data structures that are used for internal representation of queries. A *query tree* is used to represent a *relational algebra* or extended relational algebra expression, whereas a *query graph* is used to represent a *relational calculus expression*. Then, in Section 19.1.2, we show how heuristic optimization rules are applied to convert an initial query tree into an **equivalent query tree**, which represents a different relational algebra expression that is more efficient to execute but gives the same result as the original tree. We also discuss the equivalence of various relational algebra expressions. Finally, Section 19.1.3 discusses the generation of query execution plans.

### 19.1.1 Notation for Query Trees and Query Graphs

A **query tree** is a tree data structure that corresponds to an extended relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and it represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root node*, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query.

**Figure 19.1**

Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

Figure 19.1(a) shows a query tree (the same as shown in Figure 6.9) for query Q2 in Chapters 6 to 8: For every project located in ‘Stafford’, retrieve the project number, the controlling department number, and the department manager’s last name, address, and birthdate. This query is specified on the COMPANY relational schema in Figure 5.5 and corresponds to the following relational algebra expression:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation}='Stafford'}(\text{PROJECT}))) \bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$



This corresponds to the following SQL query:

```
Q2:  SELECT  P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
      FROM    PROJECT P, DEPARTMENT D, EMPLOYEE E
      WHERE   P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
             P.Plocation= 'Stafford';
```

In Figure 19.1(a), the leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE, respectively, and the internal tree nodes represent the *relational algebra operations* of the expression. When this query tree is executed, the node marked (1) in Figure 19.1(a) must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

As we can see, the query tree represents a specific order of operations for executing a query. A more neutral data structure for representation of a query is the **query graph** notation. Figure 19.1(c) (the same as shown in Figure 6.13) shows the query graph for query Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges**, as shown in Figure 19.1(c). Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

The query graph representation does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query.<sup>2</sup> Although some optimization techniques were based on query graphs such as those originally in the INGRES DBMS, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

### 19.1.2 Heuristic Optimization of Query Trees

In general, many different relational algebra expressions—and hence many different query trees—can be **semantically equivalent**; that is, they can represent the *same query and produce the same results*.<sup>3</sup>

The query parser will typically generate a standard **initial query tree** to correspond to an SQL query, without doing any optimization. For example, for a SELECT-PROJECT-JOIN query, such as Q2, the initial tree is shown in Figure 19.1(b). The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by

<sup>2</sup>Hence, a query graph corresponds to a *relational calculus* expression as shown in Section 8.6.5.

<sup>3</sup>The same query may also be stated in various ways in a high-level query language such as SQL (see Chapters 7 and 8).

the projection on the SELECT clause attributes. Such a **canonical query tree** represents a relational algebra expression that is *very inefficient if executed directly*, because of the CARTESIAN PRODUCT ( $\times$ ) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. However, this canonical query tree in Figure 19.1(b) is in a simple standard form that can be easily created from the SQL query. It will never be executed. The heuristic query optimizer will transform this initial query tree into an equivalent **final query tree** that is efficient to execute.

The optimizer must include rules for *equivalence among extended relational algebra expressions* that can be applied to transform the initial tree into the final, optimized query tree. First we discuss informally how a query tree is transformed by using heuristics, and then we discuss general transformation rules and show how they can be used in an algebraic heuristic optimizer.

**Example of Transforming a Query.** Consider the following query Q on the database in Figure 5.5: *Find the last names of employees born after 1957 who work on a project named 'Aquarius'.* This query can be specified in SQL as follows:

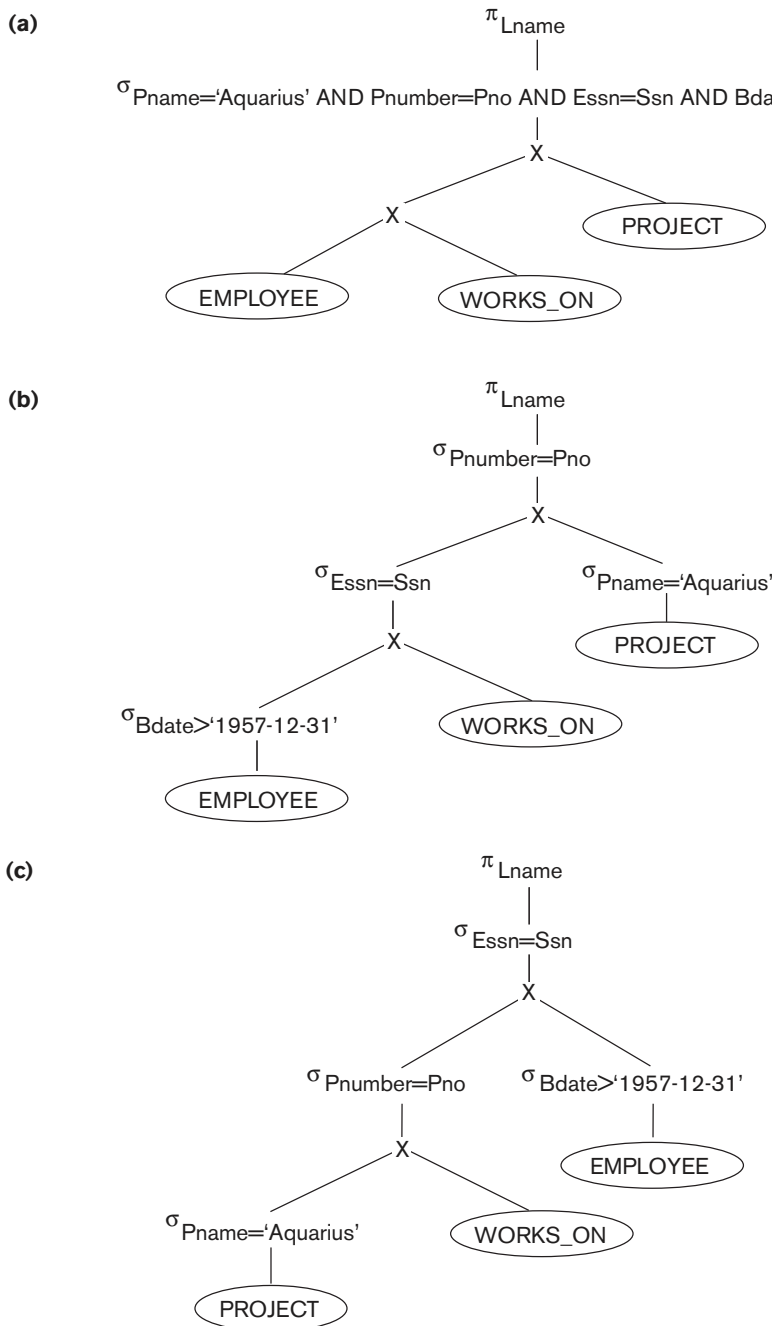
```
Q:  SELECT  E.Lname
      FROM    EMPLOYEE E, WORKS_ON W, PROJECT P
      WHERE   P.Pname='Aquarius' AND P.Pnumber=W.Pno AND E.Essn=W.Ssn
            AND E.Bdate > '1957-12-31';
```

The initial query tree for Q is shown in Figure 19.2(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS\_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree that is efficient to execute. This particular query needs only one record from the PROJECT relation—for the 'Aquarius' project—and only the EMPLOYEE records for those whose date of birth is after '1957-12-31'. Figure 19.2(b) shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure 19.2(c). This uses the information that Pnumber is a key attribute of the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only. We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition as a selection with a JOIN operation, as shown in Figure 19.2(d). Another improvement is to keep only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT ( $\pi$ ) operations as early as possible in the query tree, as shown in Figure 19.2(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

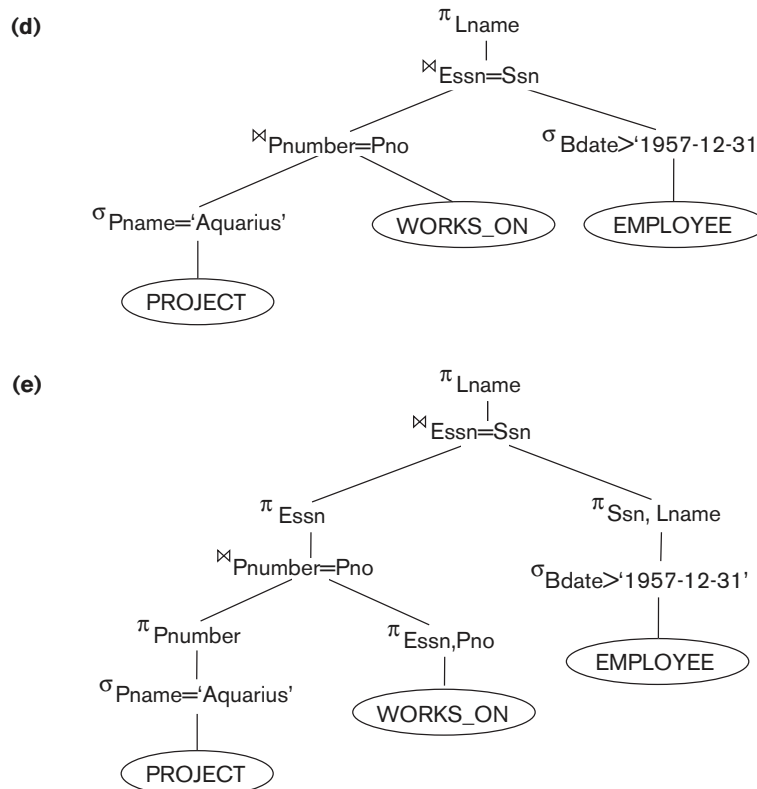
**Figure 19.2**

Steps in converting a query tree during heuristic optimization. (a) Initial (canonical) query tree for SQL query Q. (b) Moving SELECT operations down the query tree. (c) Applying the more restrictive SELECT operation first.



**Figure 19.2 (continued)**

Steps in converting a query tree during heuristic optimization. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations. (e) Moving PROJECT operations down the query tree.



As the preceding example demonstrates, a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules *preserve this equivalence*. We discuss some of these transformation rules next.

**General Transformation Rules for Relational Algebra Operations.** There are many rules for transforming relational algebra operations into equivalent ones. For query optimization purposes, we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a *different order* but the two relations represent the same information, we consider the relations to be equivalent. In Section 5.1.2 we gave an alternative definition of *relation* that makes the order of attributes unimportant; we will use this

definition here. We will state some transformation rules that are useful in query optimization, without proving them:

1. **Cascade of  $\sigma$ .** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual  $\sigma$  operations:

$$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

2. **Commutativity of  $\sigma$ .** The  $\sigma$  operation is commutative:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

3. **Cascade of  $\pi$ .** In a cascade (sequence) of  $\pi$  operations, all but the last one can be ignored:

$$\pi_{\text{List}_1}(\pi_{\text{List}_2}(\dots(\pi_{\text{List}_n}(R))\dots)) \equiv \pi_{\text{List}_1}(R)$$

4. **Commuting  $\sigma$  with  $\pi$ .** If the selection condition  $c$  involves only those attributes  $A_1, \dots, A_n$  in the projection list, the two operations can be commuted:

$$\pi_{A_1, A_2, \dots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \dots, A_n}(R))$$

5. **Commutativity of  $\bowtie$  (and  $\times$ ).** The join operation is commutative, as is the  $\times$  operation:

$$R \bowtie_c S \equiv S \bowtie_c R$$

$$R \times S \equiv S \times R$$

Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

6. **Commuting  $\sigma$  with  $\bowtie$  (or  $\times$ ).** If all the attributes in the selection condition  $c$  involve only the attributes of one of the relations being joined—say,  $R$ —the two operations can be commuted as follows:

$$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$$

Alternatively, if the selection condition  $c$  can be written as  $(c_1 \text{ AND } c_2)$ , where condition  $c_1$  involves only the attributes of  $R$  and condition  $c_2$  involves only the attributes of  $S$ , the operations commute as follows:

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$$

The same rules apply if the  $\bowtie$  is replaced by a  $\times$  operation.

7. **Commuting  $\pi$  with  $\bowtie$  (or  $\times$ ).** Suppose that the projection list is  $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$ , where  $A_1, \dots, A_n$  are attributes of  $R$  and  $B_1, \dots, B_m$  are attributes of  $S$ . If the join condition  $c$  involves only attributes in  $L$ , the two operations can be commuted as follows:

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1, \dots, A_n}(R)) \bowtie_c (\pi_{B_1, \dots, B_m}(S))$$

If the join condition  $c$  contains additional attributes not in  $L$ , these must be added to the projection list, and a final  $\pi$  operation is needed. For example, if attributes

$A_{n+1}, \dots, A_{n+k}$  of  $R$  and  $B_{m+1}, \dots, B_{m+p}$  of  $S$  are involved in the join condition  $c$  but are not in the projection list  $L$ , the operations commute as follows:

$$\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A_1, \dots, A_n, A_{n+1}, \dots, A_{n+k}}(R)) \bowtie_c (\pi_{B_1, \dots, B_m, B_{m+1}, \dots, B_{m+p}}(S)))$$

For  $\times$ , there is no condition  $c$ , so the first transformation rule always applies by replacing  $\bowtie_c$  with  $\times$ .

8. **Commutativity of set operations.** The set operations  $\cup$  and  $\cap$  are commutative, but  $-$  is not.
9. **Associativity of  $\bowtie$ ,  $\times$ ,  $\cup$ , and  $\cap$ .** These four operations are individually associative; that is, if both occurrences of  $\theta$  stand for the same operation that is any one of these four operations (throughout the expression), we have:

$$(R \theta S) \theta T \equiv R \theta (S \theta T)$$

10. **Commuting  $\sigma$  with set operations.** The  $\sigma$  operation commutes with  $\cup$ ,  $\cap$ , and  $-$ . If  $\theta$  stands for any one of these three operations (throughout the expression), we have:

$$\sigma_c (R \theta S) \equiv (\sigma_c (R)) \theta (\sigma_c (S))$$

11. **The  $\pi$  operation commutes with  $\cup$ .**

$$\pi_L (R \cup S) \equiv (\pi_L (R)) \cup (\pi_L (S))$$

12. **Converting a  $(\sigma, \times)$  sequence into  $\bowtie$ .** If the condition  $c$  of a  $\sigma$  that follows a  $\times$  corresponds to a join condition, convert the  $(\sigma, \times)$  sequence into a  $\bowtie$  as follows:

$$(\sigma_c (R \times S)) \equiv (R \bowtie_c S)$$

13. **Pushing  $\sigma$  in conjunction with set difference.**

$$\sigma_c (R - S) = \sigma_c (R) - \sigma_c (S)$$

However,  $\sigma$  may be applied to only one relation:

$$\sigma_c (R - S) = \sigma_c (R) - S$$

14. **Pushing  $\sigma$  to only one argument in  $\cap$ .**

If in the condition  $\sigma_c$  all attributes are from relation  $R$ , then:

$$\sigma_c (R \cap S) = \sigma_c (R) \cap S$$

15. **Some trivial transformations.**

If  $S$  is empty, then  $R \cup S = R$

If the condition  $c$  in  $\sigma_c$  is true for the entire  $R$ , then  $\sigma_c (R) = R$ .

There are other possible transformations. For example, a selection or join condition  $c$  can be converted into an equivalent condition by using the following standard rules from Boolean algebra (De Morgan's laws):

$$\text{NOT } (c_1 \text{ AND } c_2) \equiv (\text{NOT } c_1) \text{ OR } (\text{NOT } c_2)$$

$$\text{NOT } (c_1 \text{ OR } c_2) \equiv (\text{NOT } c_1) \text{ AND } (\text{NOT } c_2)$$

Additional transformations discussed in Chapters 4, 5, and 6 are not repeated here. We discuss next how transformations can be used in heuristic optimization.

**Outline of a Heuristic Algebraic Optimization Algorithm.** We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example in Figure 19.2. The steps of the algorithm are as follows:

1. Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.
2. Using Rules 2, 4, 6, and 10, 13, 14 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condition involves attributes from *only one table*, which means that it represents a *selection condition*, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from *two tables*, which means that it represents a *join condition*, the condition is moved to a location down the tree after the two tables are combined.
3. Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of *most restrictive* SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size.<sup>4</sup> Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.<sup>5</sup>
4. Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.
5. Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.

---

<sup>4</sup>Either definition can be used, since these rules are heuristic.

<sup>5</sup>Note that a CARTESIAN PRODUCT is acceptable in some cases—for example, if each relation has only a single tuple because each had a previous select condition on a key field.

6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

In our example, Figure 19.2(b) shows the tree in Figure 19.2(a) after applying steps 1 and 2 of the algorithm; Figure 19.2(c) shows the tree after step 3; Figure 19.2(d) after step 4; and Figure 19.2(e) after step 5. In step 6, we may group together the operations in the subtree whose root is the operation  $\pi_{\text{Essn}}$  into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation  $\pi_{\text{Essn}}$ , because the first grouping means that this subtree is executed first.

**Summary of Heuristics for Algebraic Optimization.** The main heuristic is to apply first the operations that reduce the size of intermediate results. This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible. Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

## 19.2 Choice of Query Execution Plans

### 19.2.1 Alternatives for Query Evaluation

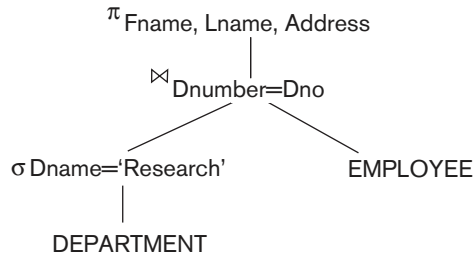
An execution plan for a relational algebra expression represented as a query tree includes information about the access methods available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree. As a simple example, consider query Q1 from Chapter 7, whose corresponding relational algebra expression is

$$\pi_{\text{Fname, Lname, Address}}(\sigma_{\text{Dname}=\text{'Research'}}(\text{DEPARTMENT}) \bowtie_{\text{Dnumber}=\text{Dno}} \text{EMPLOYEE})$$

The query tree is shown in Figure 19.3. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT (assuming one exists), an index-based nested-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE), and a scan of the JOIN result for input to the PROJECT operator. Additionally, the approach taken for executing the query may specify a materialized or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible.

With **materialized evaluation**, the result of an operation is stored as a temporary relation (that is, the result is *physically materialized*). For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which



**Figure 19.3**

A query tree for query Q1.

would produce the query result table. On the other hand, with **pipelined evaluation**, as the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence. We discussed pipelining as a strategy for query processing in Section 18.7. For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm then consumes the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm. The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.

We discussed in Section 19.1 the possibility of converting query trees into equivalent trees so that the evaluation of the query is more efficient in terms of its execution time and overall resources consumed. There are more elaborate transformations of queries that are possible to optimize, or rather to “improve.” Transformations can be applied either in a heuristic-based or cost-based manner.

As we discussed in Sections 7.1.2 and 7.1.3, nested subqueries may occur in the WHERE clause as well as in the FROM clause of SQL queries. In the WHERE clause, if an inner block makes a reference to the relation used in the outer block, it is called a correlated nested query. When a query is used within the FROM clause to define a resulting or derived relation, which participates as a relation in the outer query, it is equivalent to a view. Both these types of nested subqueries are handled by the optimizer, which transforms them and rewrites the entire query. In the next two subsections, we consider these two variations of query transformation and rewriting with examples. We will call them nested subquery optimization and subquery (view) merging transformation. In Section 19.8, we revisit this topic in the context of data warehouses and illustrate star transformation optimizations.

### 19.2.2 Nested Subquery Optimization

We discussed nested queries in Section 7.1.2. Consider the query:

```

SELECT E1.Fname, E1.Lname
FROM EMPLOYEE E1
WHERE E1.Salary = ( SELECT MAX (Salary)
                   FROM EMPLOYEE E2)
  
```

In the above nested query, there is a query block inside an outer query block. Evaluation of this query involves executing the nested query first, which yields a single value of the maximum salary  $M$  in the `EMPLOYEE` relation; then the outer block is simply executed with the selection condition `Salary = M`. The maximum salary could be obtained just from the highest value in the index on salary (if one exists) or from the catalog if it is up-to-date. The outer query is evaluated based on the same index. If no index exists, then linear search would be needed for both.

We discussed correlated nested SQL queries in Section 7.1.3. In a correlated subquery, the inner query contains a reference to the outer query via one or more variables. The subquery acts as a function that returns a set of values for each value of this variable or combination of variables.

Suppose in the database of Figure 5.5, we modify the `DEPARTMENT` relation as:

```
DEPARTMENT (Dnumber, Dname, Mgr_ssn, Mgr_start_date, Zipcode)
```

Consider the query:

```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E
WHERE EXISTS ( SELECT *
                FROM DEPARTMENT D
                WHERE D.Dnumber = E.Dno AND D.Zipcode=30332);
```

In the above, the nested subquery takes the `E.Dno`, the department where the employee works, as a parameter and returns a true or false value as a function depending on whether the department is located in zip code 30332. The naïve strategy for evaluating the query is to evaluate the inner nested subquery for every tuple of the outer relation, which is inefficient. Wherever possible, SQL optimizer tries to convert queries with nested subqueries into a join operation. The join can then be evaluated with one of the options we considered in Section 18.4. The above query would be converted to

```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E, DEPARTMENT D
WHERE D.Dnumber = E.Dno AND D.Zipcode=30332
```

The process of removing the nested query and converting the outer and inner query into one block is called **unnesting**. Here inner join is used, since `D.Dnumber` is unique and the join is an equi-join; this guarantees that a tuple from relation `Employee` will match with at most one tuple from relation `Department`. We showed in Chapter 7 that the query Q16, which has a subquery connected with the `IN` connector, was also unnested into a single block query involving a join. In general, queries involving a nested subquery connected by `IN` or `ANY` connector in SQL can always be converted into a single block query. Other techniques used include creation of temporary result tables from subqueries and using them in joins.

We repeat the example query shown in Section 18.1. (Note that the IN operator is equivalent to the =ANY operator.):

```
Q (SJ) :
SELECT COUNT(*)
FROM DEPARTMENT D
WHERE D.Dnumber IN ( SELECT E.Dno
                     FROM EMPLOYEE E
                     WHERE E.Salary > 200000)
```

In this case again, there are two options for the optimizer:

1. Evaluate the nested subquery for each outer tuple; it is inefficient to do so.
2. Unnest the subquery using **semi-join**, which is much more efficient than option 1. In Section 18.1, we used this alternative to introduce and define the semi-join operator. Note that for unnesting this subquery, which refers to expressing it as a single block, inner join *cannot* be used, since in inner join a tuple of DEPARTMENT may match more than one tuple of EMPLOYEE and thus produce wrong results. It is easy to see that a nested subquery acts as a **filter** and thus it cannot, unlike inner join, produce more rows than there are in the DEPARTMENT table. Semi-join simulates this behavior.

The process we described as **unnesting** is sometimes called **decorrelation**. We showed another example in Section 18.1 using the connector “NOT IN”, which was converted into a single block query using the operation **anti-join**. Optimization of complex nested subqueries is difficult and requires techniques that can be quite involved. We illustrate two such techniques in Section 19.2.3 below. Unnesting is a powerful optimization technique and is used widely by SQL optimizers.

### 19.2.3 Subquery (View) Merging Transformation

There are instances where a subquery appears in the FROM clause of a query and amounts to including a derived relation, which is similar to a predefined view that is involved in the query. This FROM clause subquery is often referred to as an inline view. Sometimes, an actual view defined earlier as a separate query is used as one of the argument relations in a new query. In such cases, the transformation of the query can be referred to as a view-merging or subquery merging transformation. The techniques of view merging discussed here apply equally to both inline and predefined views,

Consider the following three relations:

```
EMP (Ssn, Fn, Ln, Dno)
DEPT (Dno, Dname, Dmgrname, Bldg_id)
BLDG (Bldg_id, No_storeys, Addr, Phone)
```

The meaning of the relations is self-explanatory; the last one represents buildings where departments are located; the phone refers to a phone number for the building lobby.

The following query uses an inline view in the FROM clause; it retrieves for employees named “John” the last name, address and phone number of building where they work:

```
SELECT E.Ln, V.Addr, V.Phone
FROM EMP E, ( SELECT D.Dno, D.Dname, B.Addr, B.Phone
              FROM DEPT D, BLDG B
              WHERE D.Bldg_id = B.Bldg_id ) V
WHERE V.Dno = E.Dno AND E.Fn = “John”;
```

The above query joins the EMP table with a view called V that provides the address and phone of the building where the employee works. In turn, the view joins the two tables DEPT and BLDG. This query may be executed by first temporarily materializing the view and then joining it with the EMP table. The optimizer is then constrained to consider the join order E, V or V, E; and for computing the view, the join orders possible are D, B and B, D. Thus the total number of join order candidates is limited to 4. Also, index-based join on E, V is precluded since there is no index on V on the join column Dno. The **view-merging** operation merges the tables in the view with the tables from the outer query block and produces the following query:

```
SELECT E.Ln, B.Addr, B.Phone
FROM EMP E, DEPT D, BLDG B
WHERE D.Bldg_id = B.Bldg_id AND D.Dno = E.Dno AND E.Fn = “John”;
```

With the merged query block above, three tables appear in the FROM clause, thus affording eight possible join orders and indexes on Dno in DEPT, and Bldg\_id in BLDG can be used for index-based nested loop joins that were previously excluded. We leave it to the reader to develop execution plans with and without merging to see the comparison.

In general, views containing select-project-join operations are considered simple views and they can always be subjected to this type of view-merging. Typically, view merging enables additional options to be considered and results in an execution plan that is better than one without view merging. Sometimes other optimizations are enabled, such as dropping a table in the outer query if it is used within the view. View-merging may be invalid under certain conditions where the view is more complex and involves DISTINCT, OUTER JOIN, AGGREGATION, GROUP BY set operations, and so forth. We next consider a possible situation of GROUP-BY view-merging.

**GROUP-BY View-Merging:** When the view has additional constructs besides select-project-join as we mentioned above, merging of the view as shown above may or may not be desirable. Delaying the Group By operation after performing joins may afford the advantage of reducing the data subjected to grouping in case the joins have low join selectivity. Alternately, performing early Group By may be advantageous by reducing the amount of data subjected to subsequent joins. The optimizer would typically consider execution plans with and without merging and

compare their cost to determine the viability of doing the merging. We illustrate with an example.

Consider the following relations:

```
SALES (Custid, Productid, Date, Qty_sold)
CUST (Custid, Custname, Country, Cemail)
PRODUCT (Productid, Pname, Qty_onhand)
```

The query: List customers from France who have bought more than 50 units of a product “Ring\_234” may be set up as follows:

A view is created to count total quantity of any item bought for the <Custid, Productid> pairs:

```
CREATE VIEW CP_BOUGHT_VIEW AS
SELECT SUM (S.Qty_sold) as Bought, S.Custid, S.Productid
FROM SALES S
GROUP BY S.Custid, S.Productid;
```

Then the query using this view becomes:

```
QG: SELECT C.Custid, C.Custname, C.Cemail
FROM CUST C, PRODUCT P, CP_BOUGHT_VIEW V1
WHERE P.Productid = V1.Productid AND C.Custid = V1.Custid AND V1.
Bought > 50
AND Pname = “Ring_234” AND C.Country = “France”;
```

The view V1 may be evaluated first and its results temporarily materialized, then the query QG may be evaluated using the materialized view as one of the tables in the join. By using the merging transformation, this query becomes:

```
QT: SELECT C.Custid, C.Custname, C.Cemail
FROM CUST C, PRODUCT P, SALES S
WHERE P.Productid = S.Productid AND C.Custid = S.Custid AND
Pname = “Ring_234” AND C.Country = “France”
GROUP BY, P.Productid, P.rowid, C.rowid, C.Custid, C.Custname, C.Cemail
HAVING SUM (S.Qty_sold) > 50;
```

After merging, the resulting query QT is much more efficient and cheaper to execute. The reasoning is as follows. Before merging, the view V1 does grouping on the entire SALES table and materializes the result, and it is expensive to do so. In the transformed query, the grouping is applied to the join of the three tables; in this operation, a single product tuple is involved from the PRODUCT table, thus filtering the data from SALES considerably. The join in QT after transformation may be slightly more expensive in that the whole SALES relation is involved rather than the aggregated view table CP\_BOUGHT\_VIEW in QG. Note, however, that the GROUP-BY operation in V1 produces a table whose cardinality is not considerably smaller than the cardinality of SALES, because the grouping is on <Custid, Productid>, which may not have high repetition in SALES. Also note the use of P.rowid and C.rowid, which refer to the unique row identifiers that are added to maintain equivalence with the original query. We reiterate that the decision to merge GROUP-BY views must be made by the optimizer based on estimated costs.

### 19.2.4 Materialized Views

We discussed the concept of views in Section 7.3 and also introduced the concept of materialization of views. A view is defined in the database as a query, and a **materialized view** stores the results of that query. Using materialized views to avoid some of the computation involved in a query is another query optimization technique. A materialized view may be stored temporarily to allow more queries to be processed against it or permanently, as is common in data warehouses (see Chapter 29). A materialized view constitutes derived data because its content can be computed as a result of processing the defining query of the materialized view. The main idea behind materialization is that it is much cheaper to read it when needed and query against it than to recompute it from scratch. The savings can be significant when the view involves costly operations like join, aggregation, and so forth.

Consider, for example, view V2 in Section 7.3, which defines the view as a relation by joining the DEPARTMENT and EMPLOYEE relations. For every department, it computes the total number of employees and the total salary paid to employees in that department. If this information is frequently required in reports or queries, this view may be permanently stored. The materialized view may contain data related only to a fragment or sub-expression of the user query. Therefore, an involved algorithm is needed to replace only the relevant fragments of the query with one or more materialized views and compute the rest of the query in a conventional way. We also mentioned in Section 7.3 three update (also known as refresh) strategies for updating the view:

- Immediate update, which updates the view as soon as any of the relations participating in the view are updated
- Lazy update, which recomputes the view only upon demand
- Periodic update (or deferred update), which updates the view later, possibly with some regular frequency

When immediate update is in force, it constitutes a large amount of overhead to keep the view updated when any of the underlying base relations have a change in the form of insert, delete, and modify. For example, deleting an employee from the database, or changing the salary of an employee, or hiring a new employee affects the tuple corresponding to that department in the view and hence would require the view V2 in Section 7.3 to be immediately updated. These updates are handled sometimes manually by programs that update all views defined on top of a base relation whenever the base relation is updated. But there is obviously no guarantee that all views may be accounted for. Triggers (see Section 7.2) that are activated upon an update to the base relation may be used to take action and make appropriate changes to the materialized views. The straightforward and naive approach is to recompute the entire view for every update to any base table and is prohibitively costly. Hence incremental view maintenance is done in most RDBMSs today. We discuss that next.

**Incremental View Maintenance.** The basic idea behind incremental view maintenance is that instead of creating the view from scratch, it can be updated incrementally

by accounting for only the changes that occurred since the last time it was created/updated. The trick is in figuring out exactly what is the net change to the materialized view based on a set of inserted or deleted tuples in the base relation. We describe below the general approaches to incremental view maintenance for views involving join, selection, projection, and a few types of aggregation. To deal with modification, we can consider these approaches as a combination of delete of the old tuple followed by an insert of the new tuple. Assume a view  $V$  defined over relations  $R$  and  $S$ . The respective instances are  $v$ ,  $r$ , and  $s$ .

**Join:** If a view contains inner join of relations  $r$  and  $s$ ,  $v_{old} = r \bowtie s$ , and there is a new set of tuples inserted:  $r_i$  in  $r$ , then the new value of the view contains  $(r \cup r_i) \bowtie s$ . The incremental change to the view can be computed as  $v_{new} = r \bowtie s \cup r_i \bowtie s$ . Similarly, deleting a set of tuples  $r_d$  from  $r$  results in the new view as  $v_{new} = r \bowtie s - r_d \bowtie s$ . We will have similar expressions symmetrically when  $s$  undergoes addition or deletion.

**Selection:** If a view is defined as  $V = \sigma_C R$  with condition  $C$  for selection, when a set of tuples  $r_i$  are inserted into  $r$ , the view can be modified as  $v_{new} = v_{old} \cup \sigma_C r_i$ . On the other hand, upon deletion of tuples  $r_d$  from  $r$ , we get  $v_{new} = v_{old} - \sigma_C r_d$ .

**Projection:** Compared to the above strategy, projection requires additional work. Consider the view defined as  $V = \pi_{Sex, Salary} R$ , where  $R$  is the EMPLOYEE relation, and suppose the following  $\langle Sex, Salary \rangle$  pairs exist for Salary of 50,000 in  $r$  in three distinct tuples:  $t_5$  contains  $\langle M, 50000 \rangle$ ,  $t_{17}$  contains  $\langle M, 50000 \rangle$  and  $t_{23}$  contains  $\langle F, 50000 \rangle$ . The view  $v$  therefore contains  $\langle M, 50000 \rangle$  and  $\langle F, 50000 \rangle$  as two tuples derived from the three tuples of  $r$ . If tuple  $t_5$  were to be deleted from  $r$ , it would have no effect on the view. However, if  $t_{23}$  were to be deleted from  $r$ , the tuple  $\langle F, 50000 \rangle$  would have to be removed from the view. Similarly, if another new tuple  $t_{77}$  containing  $\langle M, 50000 \rangle$  were to be inserted in the relation  $r$ , it also would have no effect on the view. Thus, view maintenance of projection views requires a count to be maintained in addition to the actual columns in the view. In the above example, the original count values are 2 for  $\langle M, 50000 \rangle$  and 1 for  $\langle F, 50000 \rangle$ . Each time an insert to the base relation results in contributing to the view, the count is incremented; if a deleted tuple from the base relation has been represented in the view, its count is decremented. When the count of a tuple in the view reaches zero, the tuple is actually dropped from the view. When a new inserted tuple contributes to the view, its count is set to 1. Note that the above discussion assumes that SELECT DISTINCT is being used in defining the view to correspond to the project ( $\pi$ ) operation. If the multiset version of projection is used with no DISTINCT, the counts would still be used. There is an option to display the view tuple as many times as its count in case the view must be displayed as a multiset.

**Intersection:** If the view is defined as  $V = R \cap S$ , when a new tuple  $r_i$  is inserted, it is compared against the  $s$  relation to see if it is present there. If present, it is inserted in  $v$ , else not. If tuple  $r_d$  is deleted, it is matched against the view  $v$  and, if present there, it is removed from the view.



**Aggregation (Group By):** For aggregation, let us consider that GROUP BY is used on column G in relation R and the view contains (SELECT G, aggregate-function (A)). The view is a result of some aggregation function applied to attribute A, which corresponds to (see Section 8.4.2):

$$G \mathcal{S}_{\text{Aggregate-function}}(A)$$

We consider a few aggregate-functions below:

- **Count:** For keeping the count of tuples for each group, if a new tuple is inserted in r, and if it has a value for G = g1, and if g1 is present in the view, then its count is incremented by 1. If there is no tuple with the value g1 in the view, then a new tuple is inserted in the view: <g1, 1>. When the tuple being deleted has the value G = g1, its count is decremented by 1. If the count of g1 reaches zero after deletion in the view, that tuple is removed from the view.
- **Sum:** Suppose the view contains (G, sum(A)). There is a count maintained for each group in the view. If a tuple is inserted in the relation r and has (g1, x1) under the columns R.G and R.A, and if the view does not have an entry for g1, a new tuple <g1, x1> is inserted in the view and its count is set to 1. If there is already an entry for g1 as <g1, s1> in the old view, it is modified as <g1, s1 + x1> and its count is incremented by 1. For the deletion from base relation of a tuple with R.G, R.A being <g1, x1>, if the count of the corresponding group g1 is 1, the tuple for group g1 would be removed from the view. If it is present and has count higher than 1, the count would be decremented by 1 and the sum s1 would be decremented to s1 - x1.
- **Average:** The aggregate function cannot be maintained by itself without maintaining the sum and the count functions and then computing the average as sum divided by count. So both the sum and count functions need to be maintained and incrementally updated as discussed above to compute the new average.
- **Max and Min:** We can just consider Max. Min would be symmetrically handled. Again for each group, the (g, max(a), count) combination is maintained, where max(a) represents the maximum value of R.A in the base relation. If the inserted tuple has R.A value lower than the current max(a) value, or if it has a value equal to max(a) in the view, only the count for the group is incremented. If it has a value greater than max(a), the max value in the view is set to the new value and the count is incremented. Upon deletion of a tuple, if its R.A value is less than the max(a), only the count is decremented. If the R.A value matches the max(a), the count is decremented by 1; so the tuple that represented the max value of A has been deleted. Therefore, a new max must be computed for A for the group that requires substantial amount of work. If the count becomes 0, that group is removed from the view because the deleted tuple was the last tuple in the group.

We discussed incremental materialization as an optimization technique for maintaining views. However, we can also look upon materialized views as a way to reduce the effort in certain queries. For example, if a query has a component, say,  $R \bowtie S$  or  $\pi_L R$  that is available as a view, then the query may be modified to use the



view and avoid doing unnecessary computation. Sometimes an opposite situation happens. A view  $V$  is used in the query  $Q$ , and that view has been materialized as  $v$ ; let us say the view includes  $R \bowtie S$ ; however, no access structures like indexes are available on  $v$ . Suppose that indexes are available on certain attributes, say,  $A$  of the component relation  $R$  and that the query  $Q$  involves a selection condition on  $A$ . In such cases, the query against the view can benefit by using the index on a component relation, and the view is replaced by its defining query; the relation representing the materialized view is not used at all.

### 19.3 Use of Selectivities in Cost-Based Optimization

A query optimizer does not depend solely on heuristic rules or query transformations; it also estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the *lowest cost estimate*. For this approach to work, accurate *cost estimates* are required so that different strategies can be compared fairly and realistically. In addition, the optimizer must limit the number of execution strategies to be considered; otherwise, too much time will be spent making cost estimates for the many possible execution strategies. Hence, this approach is more suitable for **compiled queries**, rather than ad-hoc queries where the optimization is done at compile time and the resulting execution strategy code is stored and executed directly at runtime. For **interpreted queries**, where the entire process shown in Figure 18.1 occurs at runtime, a full-scale optimization may slow down the response time. A more elaborate optimization is indicated for compiled queries, whereas a partial, less time-consuming optimization works best for interpreted queries.

This approach is generally referred to as **cost-based query optimization**.<sup>6</sup> It uses traditional optimization techniques that search the *solution space* to a problem for a solution that minimizes an objective (cost) function. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal (absolute best) one. In Section 19.3.1, we discuss the components of query execution cost. In Section 19.3.2, we discuss the type of information needed in cost functions. This information is kept in the DBMS catalog. In Section 19.3.3, we describe histograms that are used to keep details on the value distributions of important attributes.

The decision-making process during query optimization is nontrivial and has multiple challenges. We can abstract the overall cost-based query optimization approach in the following way:

- For a given subexpression in the query, there may be multiple equivalence rules that apply. The process of applying equivalences is a cascaded one; it

---

<sup>6</sup>This approach was first used in the optimizer for the SYSTEM R in an experimental DBMS developed at IBM (Selinger et al., 1979).

does not have any limit and there is no definitive convergence. It is difficult to conduct this in a space-efficient manner.

- It is necessary to resort to some quantitative measure for evaluation of alternatives. By using the space and time requirements and reducing them to some common metric called cost, it is possible to devise some methodology for optimization.
- Appropriate search strategies can be designed by keeping the cheapest alternatives and pruning the costlier alternatives.
- The scope of query optimization is generally a query block. Various table and index access paths, join permutations (orders), join methods, group-by methods, and so on provide the alternatives from which the query optimizer must chose.
- In a global query optimization, the scope of optimization is multiple query blocks.<sup>7</sup>

### 19.3.1 Cost Components for Query Execution

The cost of executing a query includes the following components:

1. **Access cost to secondary storage.** This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers. This is also known as *disk I/O (input/output) cost*. The cost of searching for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.
2. **Disk storage cost.** This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.
3. **Computation cost.** This is the cost of performing in-memory operations on the records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as *CPU (central processing unit) cost*.
4. **Memory usage cost.** This is the cost pertaining to the number of main memory buffers needed during query execution.
5. **Communication cost.** This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated. In distributed databases (see Chapter 23), it would also include the cost of transferring tables and results among various computers during query evaluation.

---

<sup>7</sup>We do not discuss global optimization in this sense in the present chapter. Details may be found in Ahmed et al. (2006).

For large databases, the main emphasis is often on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory buffers. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost. In distributed databases, where many sites are involved (see Chapter 23), communication cost must be minimized. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components. This is why some cost functions consider a single factor only—disk access. In the next section, we discuss some of the information that is needed for formulating cost functions.

### 19.3.2 Catalog Information Used in Cost Functions

To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions. This information may be stored in the DBMS catalog, where it is accessed by the query optimizer. First, we must know the size of each file. For a file whose records are all of the same type, the **number of records (tuples) ( $r$ )**, the (average) **record size ( $R$ )**, and the **number of file blocks ( $b$ )** (or close estimates of them) are needed. The **blocking factor ( $bfr$ )** for the file may also be needed. These were mentioned in Section 18.3.4, and we utilized them while illustrating the various implementation algorithms for relational operations. We must also keep track of the *primary file organization* for each file. The primary file organization records may be *unordered*, *ordered* by an attribute with or without a primary or clustering index, or *hashed* (static hashing or one of the dynamic hashing methods) on a key attribute. Information is also kept on all primary, secondary, or clustering indexes and their indexing attributes. The **number of levels ( $x$ )** of each multilevel index (primary, secondary, or clustering) is needed for cost functions that estimate the number of block accesses that occur during query execution. In some cost functions the **number of first-level index blocks ( $b_1$ )** is needed.

Another important parameter is the **number of distinct values NDV ( $A, R$ )** of an attribute in relation  $R$  and the attribute **selectivity ( $sl$ )**, which is the fraction of records satisfying an equality condition on the attribute. This allows estimation of the **selection cardinality ( $s = sl \cdot r$ )** of an attribute, which is the *average* number of records that will satisfy an equality selection condition on that attribute.

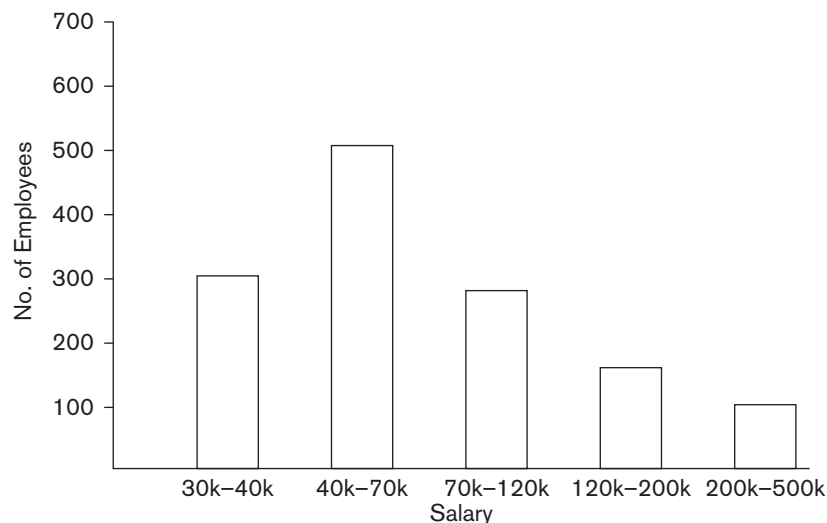
Information such as the number of index levels is easy to maintain because it does not change very often. However, other information may change frequently; for example, the number of records  $r$  in a file changes every time a record is inserted or deleted. The query optimizer will need reasonably close but not necessarily completely up-to-the-minute values of these parameters for use in estimating the cost of various execution strategies. To help with estimating the size of the results of queries, it is important to have as good an estimate of the distribution of values as possible. To that end, most systems store a histogram.

### 19.3.3 Histograms

**Histograms** are tables or data structures maintained by the DBMS to record information about the distribution of data. It is customary for most RDBMSs to store histograms for most of the important attributes. Without a histogram, the best assumption is that values of an attribute are uniformly distributed over its range from high to low. Histograms divide the attribute over important ranges (called buckets) and store the total number of records that belong to that bucket in that relation. Sometimes they may also store the number of distinct values in each bucket as well. An implicit assumption is made sometimes that among the distinct values within a bucket there is a uniform distribution. All these assumptions are oversimplifications that rarely hold. So keeping a histogram with a finer granularity (i.e., larger number of buckets) is always useful. A couple of variations of histograms are common: in **equi-width** histograms, the range of values is divided into equal subranges. In **equi-height** histograms, the buckets are so formed that each one contains roughly the same number of records. Equi-height histograms are considered better since they keep fewer numbers of more frequently occurring values in one bucket and more numbers of less frequently occurring ones in a different bucket. So the uniform distribution assumption within a bucket seems to hold better. We show an example of a histogram for salary information in a company in Figure 19.4. This histogram divides the salary range into five buckets that may correspond to the important sub-ranges over which the queries may be likely because they belong to certain types of employees. It is neither an equi-width nor an equi-height histogram.

**Figure 19.4**

Histogram of salary in the relation EMPLOYEE.



## 19.4 Cost Functions for SELECT Operation

We now provide cost functions for the selection algorithms S1 to S8 discussed in Section 18.3.1 in terms of *number of block transfers* between memory and disk. Algorithm S9 involves an intersection of record pointers after they have been retrieved by some other means, such as algorithm S6, and so the cost function will be based on the cost for S6. These cost functions are estimates that ignore computation time, storage cost, and other factors. To reiterate, the following notation is used in the formulas hereafter:

- $C_{Si}$ : Cost for method  $S_i$  in block accesses
- $r_X$ : Number of records (tuples) in a relation  $X$
- $b_X$ : Number of blocks occupied by relation  $X$  (also referred to as  $b$ )
- $bfr_X$ : Blocking factor (i.e., number of records per block) in relation  $X$
- $sl_A$ : Selectivity of an attribute  $A$  for a given condition
- $s_A$ : Selection cardinality of the attribute being selected ( $= sl_A * r$ )
- $x_A$ : Number of levels of the index for attribute  $A$
- $b_{1A}$ : Number of first-level blocks of the index on attribute  $A$
- NDV ( $A, X$ ): Number of distinct values of attribute  $A$  in relation  $X$

*Note:* In using the above notation in formulas, we have omitted the relation name or attribute name when it is obvious.

- **S1—Linear search (brute force) approach.** We search all the file blocks to retrieve all records satisfying the selection condition; hence,  $C_{S1a} = b$ . For an *equality condition on a key attribute*, only half the file blocks are searched *on the average* before finding the record, so a rough estimate for  $C_{S1b} = (b/2)$  if the record is found; if no record is found that satisfies the condition,  $C_{S1b} = b$ .
- **S2—Binary search.** This search accesses approximately  $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$  file blocks. This reduces to  $\log_2 b$  if the equality condition is on a unique (key) attribute, because  $s = 1$  in this case.
- **S3a—Using a primary index to retrieve a single record.** For a primary index, retrieve one disk block at each index level, plus one disk block from the data file. Hence, the cost is one more disk block than the number of index levels:  $C_{S3a} = x + 1$ .
- **S3b—Using a hash key to retrieve a single record.** For hashing, only one disk block needs to be accessed in most cases. The cost function is approximately  $C_{S3b} = 1$  for static hashing or linear hashing, and it is 2 disk block accesses for extendible hashing (see Section 16.8).
- **S4—Using an ordering index to retrieve multiple records.** If the comparison condition is  $>$ ,  $>=$ ,  $<$ , or  $<=$  on a key field with an ordering index, roughly half the file records will satisfy the condition. This gives a cost function of  $C_{S4} = x + (b/2)$ . This is a very rough estimate, and although it may be correct on the average, it may be inaccurate in individual cases. A more accurate estimate is possible if the distribution of records is stored in a histogram.
- **S5—Using a clustering index to retrieve multiple records.** One disk block is accessed at each index level, which gives the address of the first file disk

block in the cluster. Given an equality condition on the indexing attribute,  $s$  records will satisfy the condition, where  $s$  is the selection cardinality of the indexing attribute. This means that  $\lceil (s/bfr) \rceil$  file blocks will be in the cluster of file blocks that hold all the selected records, giving  $C_{S5} = x + \lceil (s/bfr) \rceil$ .

- **S6—Using a secondary (B<sup>+</sup>-tree) index.** For a secondary index on a key (unique) attribute, with an equality (i.e.,  $\langle \text{attribute} = \text{value} \rangle$ ) selection condition, the cost is  $x + 1$  disk block accesses. For a secondary index on a nonkey (nonunique) attribute,  $s$  records will satisfy an equality condition, where  $s$  is the selection cardinality of the indexing attribute. However, because the index is nonclustering, each of the records may reside on a different disk block, so the (worst case) cost estimate is  $C_{S6a} = x + 1 + s$ . The additional 1 is to account for the disk block that contains the record pointers after the index is searched (see Figure 17.5). For range queries, if the comparison condition is  $>$ ,  $>=$ ,  $<$ , or  $<=$  and half the file records are assumed to satisfy the condition, then (very roughly) half the first-level index blocks are accessed, plus half the file records via the index. The cost estimate for this case, approximately, is  $C_{S6b} = x + (b_{I1}/2) + (r/2)$ . The  $r/2$  factor can be refined if better selectivity estimates are available through a histogram. The latter method  $C_{S6b}$  can be very costly. For a range condition such as  $v1 < A < v2$ , the selection cardinality  $s$  must be computed from the histogram or as a default, under the uniform distribution assumption; then the cost would be computed based on whether or not  $A$  is a key or nonkey with a B<sup>+</sup>-tree index on  $A$ . (We leave this as an exercise for the reader to compute under the different conditions.)
- **S7—Conjunctive selection.** We can use either S1 or one of the methods S2 to S6 discussed above. In the latter case, we use one condition to retrieve the records and then check in the main memory buffers whether each retrieved record satisfies the remaining conditions in the conjunction. If multiple indexes exist, the search of each index can produce a set of record pointers (record ids) in the main memory buffers. The intersection of the sets of record pointers (referred to in S9) can be computed in main memory, and then the resulting records are retrieved based on their record ids.
- **S8—Conjunctive selection using a composite index.** Same as S3a, S5, or S6a, depending on the type of index.
- **S9—Selection using a bitmap index.** (See Section 17.5.2.) Depending on the nature of selection, if we can reduce the selection to a set of equality conditions, each equating the attribute with a value (e.g.,  $A = \{7, 13, 17, 55\}$ ), then a bit vector for each value is accessed which is  $r$  bits or  $r/8$  bytes long. A number of bit vectors may fit in one block. Then, if  $s$  records qualify,  $s$  blocks are accessed for the data records.
- **S10—Selection using a functional index.** (See Section 17.5.3.) This works similar to S6 except that the index is based on a function of multiple attributes; if that function is appearing in the SELECT clause, the corresponding index may be utilized.

**Cost-Based Optimization Approach.** In a query optimizer, it is common to enumerate the various possible strategies for executing a query and to estimate the costs for different strategies. An optimization technique, such as dynamic programming, may be used to find the optimal (least) cost estimate efficiently without having to consider all possible execution strategies. **Dynamic programming** is an optimization technique<sup>8</sup> in which subproblems are solved only once. This technique is applicable when a problem may be broken down into subproblems that themselves have subproblems. We will visit the dynamic programming approach when we discuss join ordering in Section 19.5.5. We do not discuss optimization algorithms here; rather, we use a simple example to illustrate how cost estimates may be used.

### 19.4.1 Example of Optimization of Selection Based on Cost Formulas:

Suppose that the EMPLOYEE file in Figure 5.5 has  $r_E = 10,000$  records stored in  $b_E = 2,000$  disk blocks with blocking factor  $bfr_E = 5$  records/block and the following access paths:

1. A clustering index on Salary, with levels  $x_{\text{Salary}} = 3$  and average selection cardinality  $s_{\text{Salary}} = 20$ . (This corresponds to a selectivity of  $sl_{\text{Salary}} = 20/10000 = 0.002$ .)
2. A secondary index on the key attribute Ssn, with  $x_{\text{Ssn}} = 4$  ( $s_{\text{Ssn}} = 1$ ,  $sl_{\text{Ssn}} = 0.0001$ ).
3. A secondary index on the nonkey attribute Dno, with  $x_{\text{Dno}} = 2$  and first-level index blocks  $b_{1\text{Dno}} = 4$ . There are  $NDV(Dno, EMPLOYEE) = 125$  distinct values for Dno, so the selectivity of Dno is  $sl_{\text{Dno}} = (1/NDV(Dno, EMPLOYEE)) = 0.008$ , and the selection cardinality is  $s_{\text{Dno}} = (r_E * sl_{\text{Dno}}) = (r_E/NDV(Dno, EMPLOYEE)) = 80$ .
4. A secondary index on Sex, with  $x_{\text{Sex}} = 1$ . There are  $NDV(Sex, EMPLOYEE) = 2$  values for the Sex attribute, so the average selection cardinality is  $s_{\text{Sex}} = (r_E/NDV(Sex, EMPLOYEE)) = 5000$ . (Note that in this case, a histogram giving the percentage of male and female employees may be useful, unless the percentages are approximately equal.)

We illustrate the use of cost functions with the following examples:

- OP1:  $\sigma_{\text{Ssn}='123456789'}(\text{EMPLOYEE})$   
 OP2:  $\sigma_{\text{Dno}>5}(\text{EMPLOYEE})$   
 OP3:  $\sigma_{\text{Dno}=5}(\text{EMPLOYEE})$   
 OP4:  $\sigma_{\text{Dno}=5 \text{ AND SALARY}>30000 \text{ AND Sex}='F'}(\text{EMPLOYEE})$

The cost of the brute force (linear search or file scan) option S1 will be estimated as  $C_{S1a} = b_E = 2000$  (for a selection on a nonkey attribute) or  $C_{S1b} = (b_E/2) = 1,000$

<sup>8</sup>For a detailed discussion of dynamic programming as a technique of optimization, the reader may consult an algorithm textbook such as Corman et al. (2003).



(average cost for a selection on a key attribute). For OP1 we can use either method S1 or method S6a; the cost estimate for S6a is  $C_{S6a} = x_{Ssn} + 1 = 4 + 1 = 5$ , and it is chosen over method S1, whose average cost is  $C_{S1b} = 1,000$ . For OP2 we can use either method S1 (with estimated cost  $C_{S1a} = 2,000$ ) or method S6b (with estimated cost  $C_{S6b} = x_{Dno} + (b_{I1Dno}/2) + (r_E/2) = 2 + (4/2) + (10,000/2) = 5,004$ ), so we choose the linear search approach for OP2. For OP3 we can use either method S1 (with estimated cost  $C_{S1a} = 2,000$ ) or method S6a (with estimated cost  $C_{S6a} = x_{Dno} + s_{Dno} = 2 + 80 = 82$ ), so we choose method S6a.

Finally, consider OP4, which has a conjunctive selection condition. We need to estimate the cost of using any one of the three components of the selection condition to retrieve the records, plus the linear search approach. The latter gives cost estimate  $C_{S1a} = 2000$ . Using the condition  $(Dno = 5)$  first gives the cost estimate  $C_{S6a} = 82$ . Using the condition  $(Salary > 30000)$  first gives a cost estimate  $C_{S4} = x_{Salary} + (b_E/2) = 3 + (2000/2) = 1003$ . Using the condition  $(Sex = 'F')$  first gives a cost estimate  $C_{S6a} = x_{Sex} + s_{Sex} = 1 + 5000 = 5001$ . The optimizer would then choose method S6a on the secondary index on Dno because it has the lowest cost estimate. The condition  $(Dno = 5)$  is used to retrieve the records, and the remaining part of the conjunctive condition  $(Salary > 30,000 \text{ AND } Sex = 'F')$  is checked for each selected record after it is retrieved into memory. Only the records that satisfy these additional conditions are included in the result of the operation. Consider the  $Dno = 5$  condition in OP3 above; Dno has 125 values and hence a  $B^+$ -tree index would be appropriate. Instead, if we had an attribute Zipcode in EMPLOYEE and if the condition were  $Zipcode = 30332$  and we had only five zip codes, bitmap indexing could be used to know what records qualify. Assuming uniform distribution,  $s_{Zipcode} = 2,000$ . This would result in a cost of 2,000 for bitmap indexing.

## 19.5 Cost Functions for the JOIN Operation

To develop reasonably accurate cost functions for JOIN operations, we must have an estimate for the size (number of tuples) of the file that results *after* the JOIN operation. This is usually kept as a ratio of the size (number of tuples) of the resulting join file to the size of the CARTESIAN PRODUCT file, if both are applied to the same input files, and it is called the **join selectivity** (*js*). If we denote the number of tuples of a relation  $R$  by  $|R|$ , we have:

$$js = |(R \bowtie_c S)| / |(R \times S)| = |(R \bowtie_c S)| / (|R| * |S|)$$

If there is no join condition  $c$ , then  $js = 1$  and the join is the same as the CARTESIAN PRODUCT. If no tuples from the relations satisfy the join condition, then  $js = 0$ . In general,  $0 \leq js \leq 1$ . For a join where the condition  $c$  is an equality comparison  $R.A = S.B$ , we get the following two special cases:

1. If  $A$  is a key of  $R$ , then  $|(R \bowtie_c S)| \leq |S|$ , so  $js \leq (1/|R|)$ . This is because each record in file  $S$  will be joined with at most one record in file  $R$ , since  $A$  is a key of  $R$ . A special case of this condition is when attribute  $B$  is a *foreign key* of  $S$  that references the *primary key*  $A$  of  $R$ . In addition, if the foreign key  $B$



has the NOT NULL constraint, then  $js = (1/|R|)$ , and the result file of the join will contain  $|S|$  records.

2. If  $B$  is a key of  $S$ , then  $|(R \bowtie_c S)| \leq |R|$ , so  $js \leq (1/|S|)$ .

Hence a **simple formula** to use for join selectivity is:

$$js = 1 / \max(\text{NDV}(A, R), \text{NDV}(B, S))$$

Having an estimate of the join selectivity for commonly occurring join conditions enables the query optimizer to estimate the size of the resulting file after the join operation, which we call **join cardinality** ( $jc$ ).

$$jc = |(R_c S)| = js * |R| * |S|.$$

We can now give some sample *approximate* cost functions for estimating the cost of some of the join algorithms given in Section 18.4. The join operations are of the form:

$$R \bowtie_{A=B} S$$

where  $A$  and  $B$  are domain-compatible attributes of  $R$  and  $S$ , respectively. Assume that  $R$  has  $b_R$  blocks and that  $S$  has  $b_S$  blocks:

- **J1—Nested-loop join.** Suppose that we use  $R$  for the outer loop; then we get the following cost function to estimate the number of block accesses for this method, assuming *three memory buffers*. We assume that the blocking factor for the resulting file is  $bfr_{RS}$  and that the join selectivity is known:

$$C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|) / bfr_{RS})$$

The last part of the formula is the cost of writing the resulting file to disk. This cost formula can be modified to take into account different numbers of memory buffers, as presented in Section 19.4. If  $n_B$  main memory buffer blocks are available to perform the join, the cost formula becomes:

$$C_{J1} = b_R + (\lceil b_R / (n_B - 2) \rceil * b_S) + ((js * |R| * |S|) / bfr_{RS})$$

- **J2—Index-based nested-loop join (using an access structure to retrieve the matching record(s)).** If an index exists for the join attribute  $B$  of  $S$  with index levels  $x_B$ , we can retrieve each record  $s$  in  $R$  and then use the index to retrieve all the matching records  $t$  from  $S$  that satisfy  $t[B] = s[A]$ . The cost depends on the type of index. For a secondary index where  $s_B$  is the selection cardinality for the join attribute  $B$  of  $S$ ,<sup>9</sup> we get:

$$C_{J2a} = b_R + (|R| * (x_B + 1 + s_B)) + ((js * |R| * |S|) / bfr_{RS})$$

For a clustering index where  $s_B$  is the selection cardinality of  $B$ , we get

$$C_{J2b} = b_R + (|R| * (x_B + (s_B / bfr_B))) + ((js * |R| * |S|) / bfr_{RS})$$

<sup>9</sup>Selection cardinality was defined as the average number of records that satisfy an equality condition on an attribute, which is the average number of records that have the same value for the attribute and hence will be joined to a single record in the other file.

For a primary index, we get

$$C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS})$$

If a **hash key** exists for one of the two join attributes—say,  $B$  of  $S$ —we get

$$C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS})$$

where  $h \geq 1$  is the average number of block accesses to retrieve a record, given its hash key value. Usually,  $h$  is estimated to be 1 for static and linear hashing and 2 for extendible hashing. This is an optimistic estimate, and typically  $h$  ranges from 1.2 to 1.5 in practical situations.

- **J3—Sort-merge join.** If the files are already sorted on the join attributes, the cost function for this method is

$$C_{J3a} = b_R + b_S + ((js * |R| * |S|)/bfr_{RS})$$

If we must sort the files, the cost of sorting must be added. We can use the formulas from Section 18.2 to estimate the sorting cost.

- **J4—Partition-hash join (or just hash join).** The records of files  $R$  and  $S$  are partitioned into smaller files. The partitioning of each file is done using the same hashing function  $h$  on the join attribute  $A$  of  $R$  (for partitioning file  $R$ ) and  $B$  of  $S$  (for partitioning file  $S$ ). As we showed in Section 18.4, the cost of this join can be approximated to:

$$C_{J4} = 3 * (b_R + b_S) + ((js * |R| * |S|)/bfr_{RS})$$

### 19.5.1 Join Selectivity and Cardinality for Semi-Join and Anti-Join

We consider these two important operations, which are used when unnesting certain queries. In Section 18.1 we showed examples of subqueries that are transformed into these operations. The goal of these operations is to avoid the unnecessary effort of doing exhaustive pairwise matching of two tables based on the join condition. Let us consider the join selectivity and cardinality of these two types of joins.

#### Semi-Join

```
SELECT COUNT(*)
FROM T1
WHERE T1.X IN (SELECT T2.Y
               FROM T2);
```

Unnesting of the query above leads to semi-join. (In the following query, the notation “ $S=$ ” for semi-join is nonstandard.)

```
SELECT COUNT(*)
FROM T1, T2
WHERE T1.X S= T2.Y;
```

The join selectivity of the semi-join above is given by:

$$js = \text{MIN}(1, \text{NDV}(Y, T2) / \text{NDV}(X, T1))$$

The join cardinality of the semi-join is given by:

$$jc = |T1| * js$$

**Anti-Join** Consider the following query:

```
SELECT COUNT (*)
FROM T1
WHERE T1.X NOT IN (SELECT T2.Y
FROM T2);
```

Unnesting of the query above leads to anti-join.<sup>10</sup> (In the following query, the notation “A=” for anti-join is nonstandard.)

```
SELECT COUNT(*)
FROM T1, T2
WHERE T1.X A= T2.Y;
```

The join selectivity of the anti-join above is given by:

$$js = 1 - \text{MIN}(1, \text{NDV}(T2.y) / \text{NDV}(T1.x))$$

The join cardinality of the anti-join is given by:

$$jc = |T1| * js$$

### 19.5.2 Example of Join Optimization Based on Cost Formulas

Suppose that we have the EMPLOYEE file described in the example in the previous section, and assume that the DEPARTMENT file in Figure 5.5 consists of  $r_D = 125$  records stored in  $b_D = 13$  disk blocks. Consider the following two join operations:

```
OP6: EMPLOYEE ⋈Dno=Dnumber DEPARTMENT
OP7: DEPARTMENT ⋈Mgr_ssn=Ssn EMPLOYEE
```

Suppose that we have a primary index on Dnumber of DEPARTMENT with  $x_{\text{Dnumber}} = 1$  level and a secondary index on Mgr\_ssn of DEPARTMENT with selection cardinality  $s_{\text{Mgr\_ssn}} = 1$  and levels  $x_{\text{Mgr\_ssn}} = 2$ . Assume that the join selectivity for OP6 is  $js_{\text{OP6}} = (1/|\text{DEPARTMENT}|) = 1/125$ <sup>11</sup> because Dnumber is a key of DEPARTMENT. Also assume that the blocking factor for the resulting join file is  $bfr_{\text{ED}} = 4$  records

<sup>10</sup>Note that in order for anti-join to be used in the NOT IN subquery, both the join attributes, T1.X and T2.Y, must have non-null values. For a detailed discussion, consult Bellamkonda et al. (2009).

<sup>11</sup>Note that this coincides with our other formula:  $= 1 / \max(\text{NDV}(\text{Dno}, \text{EMPLOYEE}), \text{NDV}(\text{Dnumber}, \text{DEPARTMENT})) = 1 / \max(125, 125) = 1/125$ .

per block. We can estimate the worst-case costs for the JOIN operation OP6 using the applicable methods J1 and J2 as follows:

1. Using method J1 with EMPLOYEE as outer loop:

$$\begin{aligned} C_{J1} &= b_E + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 2,000 + (2,000 * 13) + (((1/125) * 10,000 * 125)/4) = 30,500 \end{aligned}$$

2. Using method J1 with DEPARTMENT as outer loop:

$$\begin{aligned} C_{J1} &= b_D + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 13 + (13 * 2,000) + (((1/125) * 10,000 * 125)/4) = 28,513 \end{aligned}$$

3. Using method J2 with EMPLOYEE as outer loop:

$$\begin{aligned} C_{J2c} &= b_E + (r_E * (x_{Dnumber} + 1)) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 2,000 + (10,000 * 2) + (((1/125) * 10,000 * 125)/4) = 24,500 \end{aligned}$$

4. Using method J2 with DEPARTMENT as outer loop:

$$\begin{aligned} C_{J2a} &= b_D + (r_D * (x_{Dno} + s_{Dno})) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 13 + (125 * (2 + 80)) + (((1/125) * 10,000 * 125)/4) = 12,763 \end{aligned}$$

5. Using method J4 gives:

$$\begin{aligned} C_{J4} &= 3 * (b_D + b_E) + ((js_{OP6} * r_E * r_D)/bfr_{ED}) \\ &= 3 * (13 + 2,000) + 2,500 = 8,539 \end{aligned}$$

Case 5 has the lowest cost estimate and will be chosen. Notice that in case 2 above, if 15 memory buffer blocks (or more) were available for executing the join instead of just 3, 13 of them could be used to hold the entire DEPARTMENT relation (outer loop relation) in memory, one could be used as buffer for the result, and one would be used to hold one block at a time of the EMPLOYEE file (inner loop file), and the cost for case 2 could be drastically reduced to just  $b_E + b_D + ((js_{OP6} * r_E * r_D)/bfr_{ED})$  or 4,513, as discussed in Section 18.4. If some other number of main memory buffers was available, say  $n_B = 10$ , then the cost for case 2 would be calculated as follows, which would also give better performance than case 4:

$$\begin{aligned} C_{J1} &= b_D + (\lceil b_D / (n_B - 2) \rceil * b_E) + ((js * |R| * |S|)/bfr_{RS}) \\ &= 13 + (\lceil 13/8 \rceil * 2,000) + (((1/125) * 10,000 * 125)/4) = 28,513 \\ &= 13 + (2 * 2,000) + 2,500 = 6,513 \end{aligned}$$

As an exercise, the reader should perform a similar analysis for OP7.

### 19.5.3 Multirelation Queries and JOIN Ordering Choices

The algebraic transformation rules in Section 19.1.2 include a commutative rule and an associative rule for the join operation. With these rules, many equivalent join expressions can be produced. As a result, the number of alternative query trees grows very rapidly as the number of joins in a query increases. A query block that joins  $n$  relations will often have  $n - 1$  join operations, and hence can have a large number of different join orders. In general, for a query block that has  $n$  relations,

there are  $n!$  join orders; Cartesian products are included in this total number. Estimating the cost of every possible join tree for a query with a large number of joins will require a substantial amount of time by the query optimizer. Hence, some pruning of the possible query trees is needed. Query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees. A **left-deep join tree** is a binary tree in which the right child of each non-leaf node is always a base relation. The optimizer would choose the particular left-deep join tree with the lowest estimated cost. Two examples of left-deep trees are shown in Figure 19.5(a). (Note that the trees in Figure 19.2 are also left-deep trees.) A **right-deep join tree** is a binary tree where the left child of every leaf node is a base relation (Figure 19.5(b)).

A **bushy join tree** is a binary tree where the left or right child of an internal node may be an internal node. Figure 19.5(b) shows a right-deep join tree whereas Figure 19.5(c) shows a bushy one using four base relations. Most query optimizers consider left-deep join trees as the preferred join tree and then choose one among the  $n!$  possible join orderings, where  $n$  is the number of relations. We discuss the join ordering issue in more detail in Sections 19.5.4 and 19.5.5. The left-deep tree has exactly one shape, and the join orders for  $N$  tables in a left-deep tree are given by  $N!$ . In contrast, the shapes of a bushy tree are given by the following recurrence relation (i.e., recursive function), with  $S(n)$  defined as follows:  $S(1) = 1$ .

$$S(n) = \sum_{i=1}^{n-1} S(i) * S(n-i)$$

The above recursive equation for  $S(n)$  can be explained as follows. It states that, for  $i$  between 1 and  $N - 1$  as the number of leaves in the left subtree, those leaves may be rearranged in  $S(i)$  ways. Similarly, the remaining  $N - i$  leaves in the right subtree can be rearranged in  $S(N - i)$  ways. The number of permutations of the bushy trees is given by:

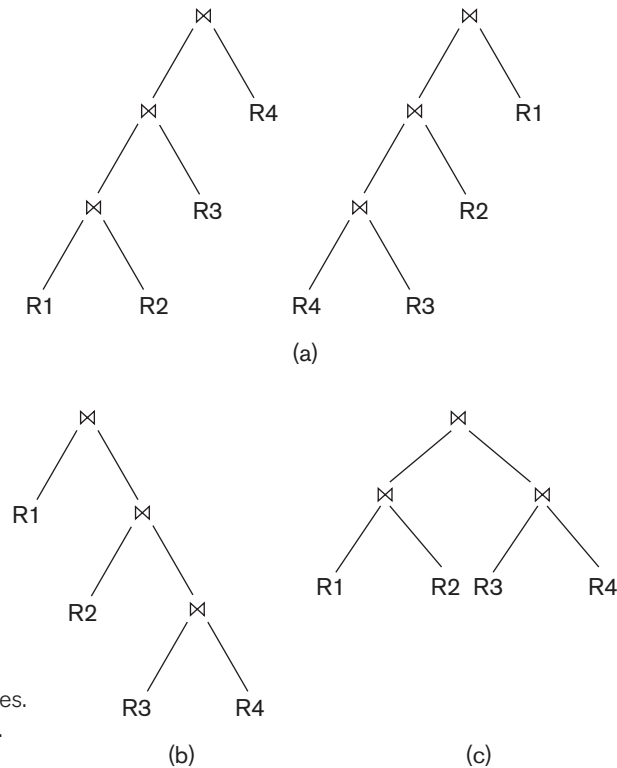
$$P(n) = n! * S(n) = (2n - 2)! / (n - 1)!$$

Table 19.1 shows the number of possible left-deep (or right-deep) join trees and bushy join trees for joins of up to seven relations.

It is clear from Table 19.1 that the possible space of alternatives becomes rapidly unmanageable if all possible bushy tree alternatives were to be considered. In certain

**Table 19.1** Number of Permutations of Left-Deep and Bushy Join Trees of  $n$  Relations

No. of Relations $N$	No. of Left-Deep Trees $N!$	No. of Bushy Shapes $S(N)$	No. of Bushy Trees $(2N - 2)! / (N - 1)!$
2	2	1	2
3	6	2	12
4	24	5	120
5	120	14	1,680
6	720	42	30,240
7	5,040	132	665,280

**Figure 19.5**

(a) Two left-deep join query trees.  
 (b) A right-deep join query tree.  
 (c) A bushy query tree.

cases like complex versions of snowflake schemas (see Section 29.3), approaches to considering bushy tree alternatives have been proposed.<sup>12</sup>

With left-deep trees, the right child is considered to be the inner relation when executing a nested-loop join, or the probing relation when executing an index-based nested-loop join. One advantage of left-deep (or right-deep) trees is that they are amenable to pipelining, as discussed in Section 18.7. For instance, consider the first left-deep tree in Figure 19.5(a) and assume that the join algorithm is the index-based nested-loop method; in this case, a disk page of tuples of the outer relation is used to probe the inner relation for matching tuples. As resulting tuples (records) are produced from the join of R1 and R2, they can be used to probe R3 to locate their matching records for joining. Likewise, as resulting tuples are produced from this join, they could be used to probe R4. Another advantage of left-deep (or right-deep) trees is that having a base relation as one of the inputs of each join allows the optimizer to utilize any access paths on that relation that may be useful in executing the join.

If materialization is used instead of pipelining (see Sections 18.7 and 19.2), the join results could be materialized and stored as temporary relations. The key idea from

<sup>12</sup>As a representative case for bushy trees, refer to Ahmed et al. (2014).

the optimizer's standpoint with respect to join ordering is to find an ordering that will reduce the size of the temporary results, since the temporary results (pipelined or materialized) are used by subsequent operators and hence affect the execution cost of those operators.

### 19.5.4 Physical Optimization

For a given logical query plan based on the heuristics we have been discussing so far, each operation needs a further decision in terms of executing the operation by a specific algorithm at the physical level. This is referred to as **physical optimization**. If this optimization is based on the relative cost of each possible implementation, we call it cost-based physical optimization. The two sets of approaches to this decision making may be broadly classified as top-down and bottom-up approaches. In the **top-down** approach, we consider the options for implementing each operation working our way down the tree and choosing the best alternative at each stage. In the **bottom-up** approach, we consider the operations working up the tree, evaluating options for physical execution, and choosing the best at each stage. Theoretically, both approaches amount to evaluation of the entire space of possible implementation solutions to minimize the cost of evaluation; however, the bottom-up strategy lends itself naturally to pipelining and hence is used in commercial RDBMSs. Among the most important physical decisions is the ordering of join operations, which we will briefly discuss in Section 19.5.5. There are certain heuristics applied at the physical optimization stage that make elaborate cost computations unnecessary. These heuristics include:

- For selections, use index scans wherever possible.
- If the selection condition is conjunctive, use the selection that results in the smallest cardinality first.
- If the relations are already sorted on the attributes being matched in a join, then prefer sort-merge join to other join methods.
- For union and intersection of more than two relations, use the associative rule; consider the relations in the ascending order of their estimated cardinalities.
- If one of the arguments in a join has an index on the join attribute, use that as the inner relation.
- If the left relation is small and the right relation is large and it has index on the joining column, then try index-based nested-loop join.
- Consider only those join orders where there are no Cartesian products or where all joins appear before Cartesian products.

The following are only some of the types of physical level heuristics used by the optimizer. If the number of relations is small (typically less than 6) and, therefore, possible implementations options are limited, then most optimizers would elect to apply a cost-based optimization approach directly rather than to explore heuristics.

### 19.5.5 Dynamic Programming Approach to Join Ordering

We saw in Section 19.5.3 that there are many possible ways to order  $n$  relations in an  $n$ -way join. Even for  $n = 5$ , which is not uncommon in practical applications, the possible permutations are 120 with left-deep trees and 1,680 with bushy trees. Since bushy trees expand the solution space tremendously, left-deep trees are generally preferred (over both bushy and right-deep trees). They have multiple advantages: First, they work well with the common algorithms for join, including nested-loop, index-based nested-loop, and other one-pass algorithms. Second, they can generate **fully pipelined plans** (i.e., plans where all joins can be evaluated using pipelining). Note that inner tables must always be materialized because in the join implementation algorithms, the entire inner table is needed to perform the matching on the join attribute. This is not possible with right-deep trees.

The common approach to evaluate possible permutations of joining relations is a greedy heuristic approach called dynamic programming. **Dynamic programming** is an optimization technique<sup>13</sup> where subproblems are solved only once, and it is applicable when a problem may be broken down into subproblems that themselves have subproblems. A typical dynamic programming algorithm has the following characteristics<sup>14</sup>:

1. The structure of an optimal solution is developed.
2. The value of the optimal solution is recursively defined.
3. The optimal solution is computed and its value developed in a bottom-up fashion.

Note that the solution developed by this procedure is an optimal solution and not the absolute optimal solution. To consider how dynamic programming may be applied to the join order selection, consider the problem of ordering a 5-way join of relations  $r_1, r_2, r_3, r_4, r_5$ . This problem has 120 ( $=5!$ ) possible left-deep tree solutions. Ideally, the cost of each of them can be estimated and compared and the best one selected. Dynamic programming takes an approach that breaks down this problem to make it more manageable. We know that for three relations, there are only six possible left-deep tree solutions. Note that if all possible bushy tree join solutions were to be evaluated, there would be 12 of them. We can therefore consider the join to be broken down as:

$$r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \bowtie r_5 = (r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

The 6 ( $=3!$ ) possible options of  $(r_1 \bowtie r_2 \bowtie r_3)$  may then be combined with the 6 possible options of taking the result of the first join, say, temp1, and then considering the next join:

$$(\text{temp1} \bowtie r_4 \bowtie r_5)$$

If we were to consider the 6 options for evaluating temp1 and, for each of them, consider the 6 options of evaluating the second join  $(\text{temp1} \bowtie r_4 \bowtie r_5)$ , the possible

<sup>13</sup>For a detailed discussion of dynamic programming as a technique of optimization, the reader may consult an algorithm textbook such as Corman et al. (2003).

<sup>14</sup>Based on Chapter 16 in Corman et al. (2003).



solution space has  $6 * 6 = 36$  alternatives. This is where dynamic programming can be used to do a sort of greedy optimization. It takes the “optimal” plan for evaluating temp1 and does *not* revisit that plan. So the solution space now reduces to only 6 options to be considered for the second join. Thus the total number of options considered becomes  $6 + 6$  instead of 120 ( $=5!$ ) in the nonheuristic exhaustive approach.

The order in which the result of the join is generated is also important for finding the best overall order of joins since for using sort-merge join with the next relation, it plays an important role. The ordering beneficial for the next join is considered an **interesting join order**. This approach was first proposed in System R at IBM Research.<sup>15</sup> Besides the join attributes of the later join, System R also included grouping attributes of a later GROUP BY or a sort order at the root of the tree among interesting sort orders. For example, in the case we discussed above, the interesting join orders for the temp1 relation will include those that match the join attribute(s) required to join with either r4 or with r5. The dynamic programming algorithm can be extended to consider best join orders for each interesting sort order. The number of subsets of  $n$  relations is  $2^n$  (for  $n = 5$  it is 32;  $n = 10$  gives 1,024, which is still manageable), and the number of interesting join orders is small. The complexity of the extended dynamic programming algorithm to determine the optimal left-deep join tree permutation has been shown to be  $O(3^n)$ .

## 19.6 Example to Illustrate Cost-Based Query Optimization

We will consider query Q2 and its query tree shown in Figure 19.1(a) to illustrate cost-based query optimization:

```

Q2:  SELECT  Pnumber, Dnum, Lname, Address, Bdate
FROM    PROJECT, DEPARTMENT, EMPLOYEE
WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn AND
           Plocation='Stafford';

```

Suppose we have the information about the relations shown in Figure 19.6. The LOW\_VALUE and HIGH\_VALUE statistics have been normalized for clarity. The tree in Figure 19.1(a) is assumed to represent the result of the algebraic heuristic optimization process and the start of cost-based optimization (in this example, we assume that the heuristic optimizer does not push the projection operations down the tree).

The first cost-based optimization to consider is join ordering. As previously mentioned, we assume the optimizer considers only left-deep trees, so the potential join orders—without CARTESIAN PRODUCT—are:

1. PROJECT  $\bowtie$  DEPARTMENT  $\bowtie$  EMPLOYEE
2. DEPARTMENT  $\bowtie$  PROJECT  $\bowtie$  EMPLOYEE

<sup>15</sup>See the classic reference in this area by Selinger et al. (1979).

## 3. DEPARTMENT ⋈ EMPLOYEE ⋈ PROJECT

## 4. EMPLOYEE ⋈ DEPARTMENT ⋈ PROJECT

Assume that the selection operation has already been applied to the PROJECT relation. If we assume a materialized approach, then a new temporary relation is created after each join operation. To examine the cost of join order (1), the first join is between PROJECT and DEPARTMENT. Both the join method and the access methods for the input relations must be determined. Since DEPARTMENT has no index according to Figure 19.6, the only available access method is a table scan (that is, a linear search). The PROJECT relation will have the selection operation performed before the join, so two options exist—table scan (linear search) or use of the PROJ\_PLOC index—so the optimizer must compare the estimated costs of these two options. The statistical information on the PROJ\_PLOC index (see Figure 19.6) shows the number of index levels  $x = 2$  (root plus leaf levels). The index is nonunique

**Figure 19.6**

Sample statistical information for relations in Q2. (a) Column information.  
(b) Table information. (c) Index information.

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

\*Blevel is the number of levels without the leaf level.

(because *Plocation* is not a key of *PROJECT*), so the optimizer assumes a uniform data distribution and estimates the number of record pointers for each *Plocation* value to be 10. This is computed from the tables in Figure 19.6 by multiplying  $\text{Selectivity} * \text{Num\_rows}$ , where *Selectivity* is estimated by  $1/\text{Num\_distinct}$ . So the cost of using the index and accessing the records is estimated to be 12 block accesses (2 for the index and 10 for the data blocks). The cost of a table scan is estimated to be 100 block accesses, so the index access is more efficient as expected.

In the materialized approach, a temporary file *TEMP1* of size 1 block is created to hold the result of the selection operation. The file size is calculated by determining the blocking factor using the formula  $\text{Num\_rows}/\text{Blocks}$ , which gives  $2,000/100$  or 20 rows per block. Hence, the 10 records selected from the *PROJECT* relation will fit into a single block. Now we can compute the estimated cost of the first join. We will consider only the nested-loop join method, where the outer relation is the temporary file, *TEMP1*, and the inner relation is *DEPARTMENT*. Since the entire *TEMP1* file fits in the available buffer space, we need to read each of the *DEPARTMENT* table's five blocks only once, so the join cost is six block accesses plus the cost of writing the temporary result file, *TEMP2*. The optimizer would have to determine the size of *TEMP2*. Since the join attribute *Dnumber* is the key for *DEPARTMENT*, any *Dnum* value from *TEMP1* will join with at most one record from *DEPARTMENT*, so the number of rows in *TEMP2* will be equal to the number of rows in *TEMP1*, which is 10. The optimizer would determine the record size for *TEMP2* and the number of blocks needed to store these 10 rows. For brevity, assume that the blocking factor for *TEMP2* is five rows per block, so a total of two blocks are needed to store *TEMP2*.

Finally, the cost of the last join must be estimated. We can use a single-loop join on *TEMP2* since in this case the index *EMP\_SSN* (see Figure 19.6) can be used to probe and locate matching records from *EMPLOYEE*. Hence, the join method would involve reading in each block of *TEMP2* and looking up each of the five *Mgr\_ssn* values using the *EMP\_SSN* index. Each index lookup would require a root access, a leaf access, and a data block access ( $x + 1$ , where the number of levels  $x$  is 2). So, 10 lookups require 30 block accesses. Adding the two block accesses for *TEMP2* gives a total of 32 block accesses for this join.

For the final projection, assume pipelining is used to produce the final result, which does not require additional block accesses, so the total cost for join order (1) is estimated as the sum of the previous costs. The optimizer would then estimate costs in a similar manner for the other three join orders and choose the one with the lowest estimate. We leave this as an exercise for the reader.

## 19.7 Additional Issues Related to Query Optimization

In this section, we will discuss a few issues of interest that we have not been able to discuss earlier.

### 19.7.1 Displaying the System's Query Execution Plan

Most commercial RDBMSs have a provision to display the execution plan produced by the query optimizer so that DBA-level personnel can view such execution plans and try to understand the decision made by the optimizer.<sup>16</sup> The common syntax is some variation of EXPLAIN <query>.

- **Oracle** uses

```
EXPLAIN PLAN FOR
<SQL Query>
```

The query may involve INSERT, DELETE, and UPDATE statements; the output goes into a table called PLAN\_TABLE. An appropriate SQL query is written to read the PLAN\_TABLE. Alternately, Oracle provides two scripts UTLXPLS.SQL and UTLXPLP.SQL to display the plan table output for serial and parallel execution, respectively.

- **IBM DB2** uses

```
EXPLAIN PLAN SELECTION [additional options] FOR <SQL-query>
```

There is no plan table. The PLAN SELECTION is a command to indicate that the explain tables should be loaded with the explanations during the plan selection phase. The same statement is also used to explain XQUERY statements.

- **SQL SERVER** uses

```
SET SHOWPLAN_TEXT ON or SET SHOWPLAN_XML ON or SET
SHOWPLAN_ALL ON
```

The above statements are used before issuing the TRANSACT-SQL, so the plan output is presented as text or XML or in a verbose form of text corresponding to the above three options.

- **PostgreSQL** uses

```
EXPLAIN [set of options] <query>.where the options include ANALYZE,
VERBOSE, COSTS, BUFFERS, TIMING, etc.
```

### 19.7.2 Size Estimation of Other Operations

In Sections 19.4 and 19.5, we discussed the SELECTION and JOIN operations and size estimation of the query result when the query involves those operations. Here we consider the size estimation of some other operations.

**Projection:** For projection of the form  $\pi_{List}(R)$  expressed as SELECT <attribute-list> FROM R, since SQL treats it as a multiset, the estimated number of tuples in the result is  $|R|$ . If the DISTINCT option is used, then size of  $\pi_A(R)$  is NDV(A, R).

---

<sup>16</sup>We have just illustrated this facility without describing the syntactic details of each system.

**Set Operations:** If the arguments for an intersection, union, or set difference are made of selections on the same relation, they can be rewritten as conjunction, disjunction, or negation, respectively. For example,  $\sigma_{c1}(R) \cap \sigma_{c2}(R)$  can be rewritten as  $\sigma_{c1 \text{ AND } c2}(R)$ ; and  $\sigma_{c1}(R) \cup \sigma_{c2}(R)$  can be rewritten as  $\sigma_{c1 \text{ OR } c2}(R)$ . The size estimation can be made based on the selectivity of conditions  $c1$  and  $c2$ . Otherwise, the estimated upper bound on the size of  $r \cap s$  is the minimum of the sizes of  $r$  and  $s$ ; the estimated upper bound on the size of  $r \cup s$  is the sum of their sizes.

**Aggregation:** The size of  $\mathcal{S}_{\text{Aggregate-function}}(A) R$  is  $\text{NDV}(G, R)$  since there is one group for each unique value of  $G$ .

**Outer Join :** the size of  $R \text{ LEFT OUTER JOIN } S$  would be  $|R \bowtie S|$  plus  $|R \text{ anti-join } S|$ . Similarly, the size of  $R \text{ FULL OUTER JOIN } S$  would be  $|r \bowtie s|$  plus  $|r \text{ anti-join } s|$  plus  $|s \text{ anti-join } r|$ . We discussed anti-join selectivity estimation in Section 19.5.1.

### 19.7.3 Plan Caching

In Chapter 2, we referred to parametric users who run the same queries or transactions repeatedly, but each time with a different set of parameters. For example, a bank teller uses an account number and some function code to check the balance in that account. To run such queries or transactions repeatedly, the query optimizer computes the best plan when the query is submitted for the first time and caches the plan for future use. This storing of the plan and reusing it is referred to as **plan caching**. When the query is resubmitted with different constants as parameters, the same plan is reused with the new parameters. It is conceivable that the plan may need to be modified under certain situations; for example, if the query involves report generation over a range of dates or range of accounts, then, depending on the amount of data involved, different strategies may apply. Under a variation called **parametric query optimization**, a query is optimized without a certain set of values for its parameters and the optimizer outputs a number of plans for different possible value sets, all of which are cached. As a query is submitted, the parameters are compared to the ones used for the various plans and the cheapest among the applicable plans is used.

### 19.7.4 Top- $k$ Results Optimization

When the output of a query is expected to be large, sometimes the user is satisfied with only the top- $k$  results based on some sort order. Some RDBMSs have a **limit  $K$  clause** to limit the result to that size. Similarly, hints may be specified to inform the optimizer to limit the generation of the result. Trying to generate the entire result and then presenting only the top- $k$  results by sorting is a naive and inefficient strategy. Among the suggested strategies, one uses generation of results in a sorted order so that it can be stopped after  $K$  tuples. Other strategies, such as introducing additional selection conditions based on the estimated highest value, have been proposed. Details are beyond our scope here. The reader may consult the bibliographic notes for details.

## 19.8 An Example of Query Optimization in Data Warehouses

In this section, we introduce another example of query transformation and rewriting as a technique for query optimization. In Section 19.2, we saw examples of query transformation and rewriting. Those examples dealt with nested subqueries and used heuristics rather than cost-based optimization. The subquery (view) merging example we showed can be considered a heuristic transformation; but the group-by view merging uses cost-based optimization as well. In this section, we consider a transformation of star-schema queries in data warehouses based on cost considerations. These queries are commonly used in data warehouse applications that follow the star schema. (See Section 29.3 for a discussion of star schemas.)

We will refer to this procedure as **star-transformation optimization**. The star schema contains a collection of tables; it gets its name because of the schema's resemblance to a star-like shape whose center contains one or more fact tables (relations) that reference multiple dimension tables (relations). The fact table contains information about the relationships (e.g., sales) among the various dimension tables (e.g., customer, part, supplier, channel, year, etc.) and measure columns (e.g., amount\_sold, etc.). Consider the representative query called QSTAR given below. Assume that D1, D2, D3 are aliases for the dimension tables DIM1, DIM2, DIM3, whose primary keys are, respectively, D1.Pk, D2.Pk, and D3.Pk. These dimensions have corresponding foreign key attributes in the fact table FACT with alias F—namely, F.Fk1, F.Fk2, F.Fk3—on which joins can be defined. The query creates a grouping on attributes D1.X, D2.Y and produces a sum of the so-called “measure” attribute (see Section 29.3) F.M from the fact table F. There are conditions on attributes A, B, C in DIM1, DIM2, DIM3, respectively:

**Query QSTAR:**

```
SELECT D1.X, D2.Y, SUM (F.M)
FROM FACT F, DIM1 D1, DIM2 D2, DIM3 D3
WHERE F.Fk1 = D1.Pk and F.Fk2 = D2.Pk and F.Fk3 = D3.Pk and
      D1.A > 5 and D2.B < 77 and D3.C = 11
GROUP BY D1.X, D2.Y
```

The fact table is generally very large in comparison with the dimension tables. QSTAR is a typical star query, and its fact table tends to be generally very large and joined with several tables of small dimension tables. The query may also contain single-table filter predicates on other columns of the dimension tables, which are generally restrictive. The combination of these filters helps to significantly reduce the data set processed from the fact table (such as D1.A > 5 in the above query). This type of query generally does grouping on columns coming from dimension tables and aggregation on measure columns coming from the fact table.

The goal of star-transformation optimization is to access only this reduced set of data from the fact table and avoid using a full table scan on it. Two types of star-transformation optimizations are possible: (A) classic star transformation, and

(B) bitmap index star transformation. Both these optimizations are performed on the basis of comparative costs of the original and the transformed queries.

#### A. Classic Star Transformation

In this optimization, a Cartesian product of the dimension tables is performed first after applying the filters (such as  $D1.A > 5$ ) to each dimension table. Note that generally there are no join predicates between dimension tables. The result of this Cartesian product is then joined with the fact table using B-tree indexes (if any) on the joining keys of the fact table.

#### B. Bitmap Index Star Transformation

The requirement with this optimization is that there must be bitmap<sup>17</sup> indexes on the fact-table joining keys referenced in the query. For example, in QSTAR, there must be bitmap indexes (see Section 17.5.2) on FACT.Fk1, FACT.Fk2, and FACT.Fk3 attributes; each bit in the bitmap corresponds to a row in the fact table. The bit is set if the key value of the attribute appears in a row of the fact table. The given query QSTAR is transformed into Q2STAR as shown below.

#### Q2STAR:

```
SELECT D1.X, D2.Y, SUM (F.M)
FROM FACT F, DIM1 D1, DIM2 D2
WHERE F.Fk1 = D1.Pk and F.Fk2 = D2.Pk and D1.A > 5 and D2.B < 77 and
      F.Fk1 IN (SELECT D1.Pk
                FROM DIM1 D1
                WHERE D1.A > 5) AND
      F.Fk2 IN (SELECT D2.Pk
                FROM DIM2 D2
                WHERE D2.B < 77) AND
      F.Fk3 IN (SELECT D3.pk
                FROM DIM3 D3
                WHERE D3.C = 11)
GROUP BY D1.X, D2.Y;
```

The bitmap star transformation adds subquery predicates corresponding to the dimension tables. Note that the subqueries introduced in Q2STAR may be looked upon as a set membership operation; for example,  $F.Fk1 \text{ IN } (5, 9, 12, 13, 29 \dots)$ .

When driven by bitmap AND and OR operations of the key values supplied by the dimension subqueries, only the relevant rows from the fact table need to be retrieved. If the filter predicates on the dimension tables and the intersection of the fact table joining each dimension table filtered out a significant subset of the fact table rows, then this optimization would prove to be much more efficient than a brute force full-table scan of the fact table.

<sup>17</sup>In some cases, the B-tree index keys can be converted into bitmaps, but we will not discuss this technique here.

The following operations are performed in Q2STAR in order to access and join the FACT table.

1. By iterating over the key values coming from a dimension subquery, the bitmaps are retrieved for a given key value from a bitmap index on the FACT table.
2. For a subquery, the bitmaps retrieved for various key values are merged (OR-ed).
3. The merged bitmaps for each dimension subqueries are AND-ed; that is, a conjunction of the joins is performed.
4. From the final bitmap, the corresponding tuple-ids for the FACT table are generated.
5. The FACT table rows are directly retrieved using the tuple-ids.

**Joining Back:** The subquery bitmap trees filter the fact table based on the filter predicates on the dimension tables; therefore, it may still be necessary to join the dimension tables back to the relevant rows in the fact table using the original join predicates. The join back of a dimension table can be avoided if the column(s) selected from the subquery are unique and the columns of the dimension table are not referenced in the SELECT and GROUP-BY clauses. Note that in Q2STAR, the table DIM3 is not joined back to the FACT table, since it is not referenced in the SELECT and GROUP-BY clauses, and DIM3.Pk is unique.

## 19.9 Overview of Query Optimization in Oracle<sup>18</sup>

This section provides a broad overview of various features in Oracle query processing, including query optimization, execution, and analytics.<sup>19</sup>

### 19.9.1 Physical Optimizer

The Oracle physical optimizer is cost based and was introduced in Oracle 7.1. The scope of the physical optimizer is a single query block. The physical optimizer examines alternative table and index access paths, operator algorithms, join orderings, join methods, parallel execution distribution methods, and so on. It chooses the execution plan with the lowest estimated cost. The estimated query cost is a relative number proportional to the expected elapsed time needed to execute the query with the given execution plan.

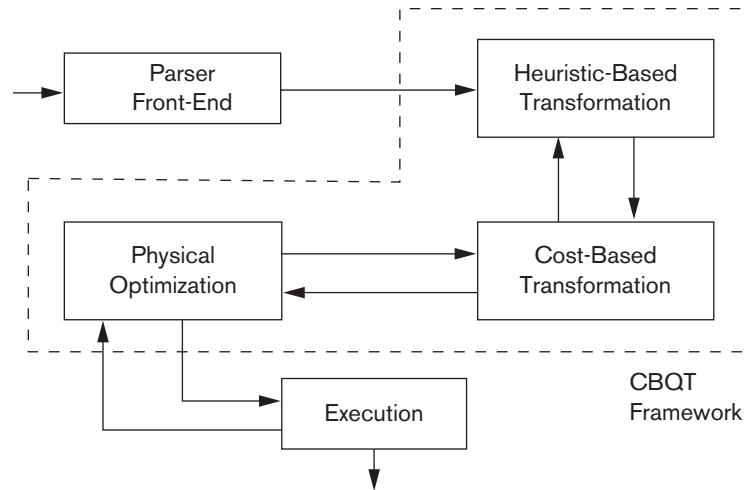
The physical optimizer calculates this cost based on object statistics (such as table cardinalities, number of distinct values in a column, column high and low values, data distribution of column values), the estimated usage of resources (such as I/O and CPU time), and memory needed. Its estimated cost is an internal metric that

---

<sup>18</sup>This section is contributed by Rafi Ahmed of Oracle Corporation.

<sup>19</sup>Support for analytics was introduced in Oracle 10.2.





**Figure 19.7**  
Cost-based query transformation framework (based on Ahmed et al., 2006).

roughly corresponds to the run time and the required resources. The goal of cost-based optimization in Oracle is to find the best trade-off between the lowest run time and the least resource utilization.

### 19.9.2 Global Query Optimizer

In traditional RDBMSs, query optimization consists of two distinct logical and physical optimization phases. In contrast, Oracle has a global query optimizer, where logical transformation and physical optimization phases have been integrated to generate an optimal execution plan for the entire query tree. The architecture of the Oracle query processing is illustrated in Figure 19.7.

Oracle performs a multitude of query transformations, which change and transform the user queries into equivalent but potentially more optimal forms. Transformations can be either heuristic-based or cost-based. The cost-based query transformation (CBQT) framework<sup>20</sup> introduced in Oracle 10g provides efficient mechanisms for exploring the state space generated by applying one or more transformations. During cost-based transformation, an SQL statement, which may comprise multiple query blocks, is copied and transformed and its cost is computed using the physical optimizer. This process is repeated multiple times, each time applying a new set of possibly interdependent transformations; and, at the end, one or more transformations are selected and applied to the original SQL statement, if those transformations result in an optimal execution plan. To deal with the combinatorial explosion, the CBQT framework provides efficient strategies for searching the state space of various transformations.

The availability of the general framework for cost-based transformation has made it possible for other innovative transformations to be added to the vast repertoire of

<sup>20</sup>As presented in Ahmed et al. (2006).

Oracle's query transformation techniques. Major among these transformations are group-by and distinct subquery merging (in the FROM clause of the query), subquery unnesting, predicate move-around, common subexpression elimination, join predicate push down, OR expansion, subquery coalescing, join factorization, subquery removal through window function, star transformation, group-by placement, and bushy join trees.<sup>21</sup>

The cost-based transformation framework of Oracle 10g is a good example of the sophisticated approach taken to optimize SQL queries.

### 19.9.3 Adaptive Optimization

Oracle's physical optimizer is adaptive and uses a feedback loop from the execution level to improve on its previous decisions. The optimizer selects the most optimal execution plan for a given SQL statement using the cost model, which relies on object statistics (e.g., number of rows, distribution of column values, etc.) and system statistics (e.g., I/O bandwidth of the storage subsystem). The optimality of the final execution plan depends primarily on the accuracy of the statistics fed into the cost model as well as on the sophistication of the cost model itself. In Oracle, the feedback loop shown in Figure 19.7 establishes a bridge between the execution engine and the physical optimizer. The bridge brings valuable statistical information to enable the physical optimizer to assess the impact of its decisions and make better decisions for the current and future executions. For example, based on the estimated value of table cardinality, the optimizer may choose the index-based nested-loop join method. However, during the execution phase, the actual table cardinality may be detected to diverge significantly from the estimated value. This information may trigger the physical optimizer to revise its decision and dynamically change the index access join method to the hash join method.

### 19.9.4 Array Processing

One of the critical deficiencies of SQL implementations is its lack of support for *N*-dimensional array-based computation. Oracle has made extensions for analytics and OLAP features; these extensions have been integrated into the Oracle RDBMS engine.<sup>22</sup> We will illustrate the need for OLAP queries when we discuss data warehousing in Chapter 29. These SQL extensions involving array-based computations for complex modeling and optimizations include access structures and execution strategies for processing these computations efficiently. The computation clause (details are beyond our scope here) allows the Oracle RDBMS to treat a table as a multidimensional array and specify a set of formulas over it. The formulas replace multiple joins and UNION operations that must be performed for equivalent computation with current ANSI SQL (where ANSI stands for

---

<sup>21</sup>More details can be found in Ahmed et al. (2006, 2014).

<sup>22</sup>See Witkowski et al. (2003) for more details.

American National Standards Institute). The computation clause not only allows for ease of application development but also offers the Oracle RDBMS an opportunity to perform better optimization.

### 19.9.5 Hints

An interesting addition to the Oracle query optimizer is the capability for an application developer to specify hints (also called query *annotations* or *directives* in other systems) to the optimizer. Hints are embedded in the text of an SQL statement. Hints are commonly used to address the infrequent cases where the optimizer chooses a suboptimal plan. The idea is that an application developer occasionally might need to override the optimizer decisions based on cost or cardinality mis-estimations. For example, consider the EMPLOYEE table shown in Figure 5.6. The Sex column of that table has only two distinct values. If there are 10,000 employees, then the optimizer, in the absence of a histogram on the Sex column, would estimate that half are male and half are female, assuming a uniform data distribution. If a secondary index exists, it would more than likely not be used. However, if the application developer knows that there are only 100 male employees, a hint could be specified in an SQL query whose WHERE-clause condition is Sex = 'M' so that the associated index would be used in processing the query. Various types of hints can be specified for different operations; these hints include but are not limited to the following:

- The access path for a given table
- The join order for a query block
- A particular join method for a join between tables
- The enabling or disabling of a transformation

### 19.9.6 Outlines

In Oracle RDBMSs, outlines are used to preserve execution plans of SQL statements or queries. Outlines are implemented and expressed as a collection of hints, because hints are easily portable and comprehensible. Oracle provides an extensive set of hints that are powerful enough to specify any execution plan, no matter how complex. When an outline is used during the optimization of an SQL statement, these hints are applied at appropriate stages by the optimizer (and other components). Every SQL statement processed by the Oracle optimizer automatically generates an outline that can be displayed with the execution plan. Outlines are used for purposes such as plan stability, what-if analysis, and performance experiments.

### 19.9.7 SQL Plan Management

Execution plans for SQL statements have a significant impact on the overall performance of a database system. New optimizer statistics, configuration parameter changes, software updates, introduction of new query optimization and processing techniques, and hardware resource utilizations are among a multitude of factors

that may cause the Oracle query optimizer to generate a new execution plan for the same SQL queries or statements. Although most of the changes in the execution plans are beneficial or benign, a few execution plans may turn out to be suboptimal, which can have a negative impact on system performance.

In Oracle 11g, a novel feature called SQL plan management (SPM) was introduced<sup>23</sup> for managing execution plans for a set of queries or workloads. SPM provides stable and optimal performance for a set of SQL statements by preventing new suboptimal plans from being executed while allowing other new plans to be executed if they are verifiably better than the previous plans. SPM encapsulates an elaborate mechanism for managing the execution plans of a set of SQL statements, for which the user has enabled SPM. SPM maintains the previous execution plans in the form of stored outlines associated with texts of SQL statements and compares the performances of the old and new execution plans for a given SQL statement before permitting them to be used by the user. SPM can be configured to work automatically, or it can be manually controlled for one or more SQL statements.

## 19.10 Semantic Query Optimization

A different approach to query optimization, called **semantic query optimization**, has been suggested. This technique, which may be used in combination with the techniques discussed previously, uses constraints specified on the database schema—such as unique attributes and other more complex constraints—to modify one query into another query that is more efficient to execute. We will not discuss this approach in detail but we will illustrate it with a simple example. Consider the SQL query:

```
SELECT    E.Lname, M.Lname
FROM      EMPLOYEE AS E, EMPLOYEE AS M
WHERE      E.Super_ssn=M.Ssn AND E.Salary > M.Salary
```

This query retrieves the names of employees who earn more than their supervisors. Suppose that we had a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it does not need to execute the query because it knows that the result of the query will be empty. This may save considerable time if the constraint checking can be done efficiently. However, searching through many constraints to find those that are applicable to a given query and that may semantically optimize it can also be time-consuming.

Consider another example:

```
SELECT Lname, Salary
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.Dno = DEPARTMENT.Dnumber and
EMPLOYEE.Salary>100000
```

---

<sup>23</sup>See Ziauddin et al. (2008).

In this example, the attributes retrieved are only from one relation: EMPLOYEE; the selection condition is also on that one relation. However, there is a referential integrity constraint that Employee.Dno is a foreign key that refers to the primary key Department.Dnumber. Therefore, this query can be transformed by removing the DEPARTMENT relation from the query and thus avoiding the inner join as follows:

```
SELECT Lname, Salary
FROM EMPLOYEE
WHERE EMPLOYEE.Dno IS NOT NULL and EMPLOYEE.Salary>100000
```

This type of transformation is based on the primary-key/foreign-key relationship semantics, which are a constraint between the two relations.

With the inclusion of active rules and additional metadata in database systems (see Chapter 26), semantic query optimization techniques are being gradually incorporated into DBMSs.

## 19.11 Summary

In the previous chapter, we presented the strategies for query processing used by relational DBMSs. We considered algorithms for various standard relational operators, including selection, projection, and join. We also discussed other types of joins, including outer join, semi-join, and anti-join, and we discussed aggregation as well as external sorting. In this chapter, our goal was to focus on query optimization techniques used by relational DBMSs. In Section 19.1 we introduced the notation for query trees and graphs and described heuristic approaches to query optimization; these approaches use heuristic rules and algebraic techniques to improve the efficiency of query execution. We showed how a query tree that represents a relational algebra expression can be heuristically optimized by reorganizing the tree nodes and transforming the tree into another equivalent query tree that is more efficient to execute. We also gave equivalence-preserving transformation rules and a systematic procedure for applying them to a query tree. In Section 19.2 we described alternative query evaluation plans, including pipelining and materialized evaluation. Then we introduced the notion of query transformation of SQL queries; this transformation optimizes nested subqueries. We also illustrated with examples of merging subqueries occurring in the FROM clause, which act as derived relations or views. We also discussed the technique of materializing views.

We discussed in some detail the cost-based approach to query optimization in Section 19.3. We discussed information maintained in catalogs that the query optimizer consults. We also discussed histograms to maintain distribution of important attributes. We showed how cost functions are developed for some database access algorithms for selection and join in Sections 19.4 and 19.5, respectively. We illustrated with an example in Section 19.6 how these cost functions are used to estimate the costs of different execution strategies. A number of additional issues such as display of query plans, size estimation of results, plan caching and top-k results optimization were discussed in Section 19.7. Section 19.8

was devoted to a discussion of how typical queries in data warehouses are optimized. We gave an example of cost-based query transformation in data warehouse queries on the so-called star schema. In Section 19.9 we presented a detailed overview of the Oracle query optimizer, which uses a number of additional techniques, details of which were beyond our scope. Finally, in Section 19.10 we mentioned the technique of semantic query optimization, which uses the semantics or integrity constraints to simplify the query or completely avoid accessing the data or the actual execution of the query.

## Review Questions

- 19.1. What is a query execution plan?
- 19.2. What is meant by the term *heuristic optimization*? Discuss the main heuristics that are applied during query optimization.
- 19.3. How does a query tree represent a relational algebra expression? What is meant by an execution of a query tree? Discuss the rules for transformation of query trees, and identify when each rule should be applied during optimization.
- 19.4. How many different join orders are there for a query that joins 10 relations? How many left-deep trees are possible?
- 19.5. What is meant by *cost-based query optimization*?
- 19.6. What is the optimization approach based on dynamic programming? How is it used during query optimization?
- 19.7. What are the problems associated with keeping views materialized?
- 19.8. What is the difference between *pipelining* and *materialization*?
- 19.9. Discuss the cost components for a cost function that is used to estimate query execution cost. Which cost components are used most often as the basis for cost functions?
- 19.10. Discuss the different types of parameters that are used in cost functions. Where is this information kept?
- 19.11. What are semi-join and anti-join? What are the join selectivity and join cardinality parameters associated with them? Provide appropriate formulas.
- 19.12. List the cost functions for the SELECT and JOIN methods discussed in Sections 19.4 and 19.5.
- 19.13. What are the special features of query optimization in Oracle that we did not discuss in the chapter?
- 19.14. What is meant by *semantic query optimization*? How does it differ from other query optimization techniques?

## Exercises

- 19.15.** Develop cost functions for the PROJECT, UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT algorithms discussed in Section 19.4.
- 19.16.** Develop cost functions for an algorithm that consists of two SELECTs, a JOIN, and a final PROJECT, in terms of the cost functions for the individual operations.
- 19.17.** Develop a pseudo-language-style algorithm for describing the dynamic programming procedure for join-order selection.
- 19.18.** Calculate the cost functions for different options of executing the JOIN operation OP7 discussed in Section 19.4.
- 19.19.** Develop formulas for the hybrid hash-join algorithm for calculating the size of the buffer for the first bucket. Develop more accurate cost estimation formulas for the algorithm.
- 19.20.** Estimate the cost of operations OP6 and OP7 using the formulas developed in Exercise 19.19.
- 19.21.** Compare the cost of two different query plans for the following query:

$$\sigma_{\text{Salary} < 40000}(\text{EMPLOYEE} \bowtie_{\text{Dno}=\text{Dnumber}} \text{DEPARTMENT})$$

Use the database statistics shown in Figure 19.6.

## Selected Bibliography

This bibliography provides literature references for the topics of query processing and optimization. We discussed query processing algorithms and strategies in the previous chapter, but it is difficult to separate the literature that addresses optimization from the literature that addresses query processing strategies and algorithms. Hence, the bibliography is consolidated.

A detailed algorithm for relational algebra optimization is given by Smith and Chang (1975). The Ph.D. thesis of Kooi (1980) provides a foundation for query processing techniques. A survey paper by Jarke and Koch (1984) gives a taxonomy of query optimization and includes a bibliography of work in this area. A survey by Graefe (1993) discusses query execution in database systems and includes an extensive bibliography.

Whang (1985) discusses query optimization in OBE (Office-By-Example), which is a system based on the language QBE. Cost-based optimization was introduced in the SYSTEM R experimental DBMS and is discussed in Astrahan et al. (1976). Selinger et al. (1979) is a classic paper that discussed cost-based optimization of multiway joins in SYSTEM R. Join algorithms are discussed in Gotlieb (1975), Blaggen and Eswaran (1976), and Whang et al. (1982). Hashing algorithms for implementing joins are described and analyzed in DeWitt et al. (1984), Bratbergsengen



(1984), Shapiro (1986), Kitsuregawa et al. (1989), and Blakeley and Martin (1990), among others. Blakeley et al. (1986) discuss maintenance of materialized views. Chaudhari et al. (1995) discuss optimization of queries with materialized views. Approaches to finding a good join order are presented in Ioannidis and Kang (1990) and in Swami and Gupta (1989). A discussion of the implications of left-deep and bushy join trees is presented in Ioannidis and Kang (1991). Kim (1982) discusses transformations of nested SQL queries into canonical representations. Optimization of aggregate functions is discussed in Klug (1982) and Muralikrishna (1992). Query optimization with Group By is presented in Chaudhari and Shim (1994). Yan and Larson (1995) discuss eager and lazy aggregation. Salzberg et al. (1990) describe a fast external sorting algorithm. Estimating the size of temporary relations is crucial for query optimization. Sampling-based estimation schemes are presented in Haas et al. (1995), Haas and Swami (1995), and Lipton et al. (1990). Having the database system store and use more detailed statistics in the form of histograms is the topic of Muralikrishna and DeWitt (1988) and Poosala et al. (1996). Galindo-Legaria and Joshi (2001) discuss nested subquery and aggregation optimization.

O'Neil and Graefe (1995) discuss multi-table joins using bitmap indexes. Kim et al. (1985) discuss advanced topics in query optimization. Semantic query optimization is discussed in King (1981) and Malley and Zdonick (1986). Work on semantic query optimization is reported in Chakravarthy et al. (1990), Shenoy and Ozsoyoglu (1989), and Siegel et al. (1992). Volcano, a query optimizer based on query equivalence rules, was developed by Graefe and McKenna (1993). Volcano and the follow-on Cascades approach by Graefe (1995) are the basis for Microsoft's SQL Server query optimization. Carey and Kossman (1998) and Bruno et al. (2002) present approaches to query optimization for top- $k$  results. Galindo Legaria et al. (2004) discuss processing and optimizing database updates.

Ahmed et al. (2006) discuss cost-based query transformation in Oracle and give a good overview of the global query optimization architecture in Oracle 10g. Ziauddin et al. (2008) discuss the idea of making the optimizer change the execution plan for a query. They discuss Oracle's SQL plan management (SPM) feature, which lends stability to performance. Bellamkonda et al. (2009) provide additional techniques for query optimization. Ahmed et al. (2014) consider the advantages of bushy trees over alternatives for execution. Witkowski et al. (2003) discuss support for  $N$ -dimensional array-based computation for analytics that has been integrated into the Oracle RDBMS engine.



This page intentionally left blank

# part 9

## **Transaction Processing, Concurrency Control, and Recovery**

This page intentionally left blank

## Introduction to Transaction Processing Concepts and Theory

The concept of *transaction* provides a mechanism for describing logical units of database processing. **Transaction processing systems** are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems require high availability and fast response time for hundreds of concurrent users. In this chapter, we present the concepts that are needed in transaction processing systems. We define the concept of a transaction, which is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness. A transaction is typically implemented by a computer program that includes database commands such as retrievals, insertions, deletions, and updates. We introduced some of the basic techniques for database programming in Chapters 10 and 11.

In this chapter, we focus on the basic concepts and theory that are needed to ensure the correct executions of transactions. We discuss the concurrency control problem, which occurs when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results. We also discuss the problems that can occur when transactions fail, and how the database system can recover from various types of failures.

This chapter is organized as follows. Section 20.1 informally discusses why concurrency control and recovery are necessary in a database system. Section 20.2 defines the term *transaction* and discusses additional concepts related to transaction

processing in database systems. Section 20.3 presents the important properties of atomicity, consistency preservation, isolation, and durability or permanency—called the ACID properties—that are considered desirable in transaction processing systems. Section 20.4 introduces the concept of schedules (or histories) of executing transactions and characterizes the *recoverability* of schedules. Section 20.5 discusses the notion of *serializability* of concurrent transaction execution, which can be used to define correct execution sequences (or schedules) of concurrent transactions. In Section 20.6, we present some of the commands that support the transaction concept in SQL, and we introduce the concepts of isolation levels. Section 20.7 summarizes the chapter.

The two following chapters continue with more details on the actual methods and techniques used to support transaction processing. Chapter 21 gives an overview of the basic concurrency control protocols and Chapter 22 introduces recovery techniques.

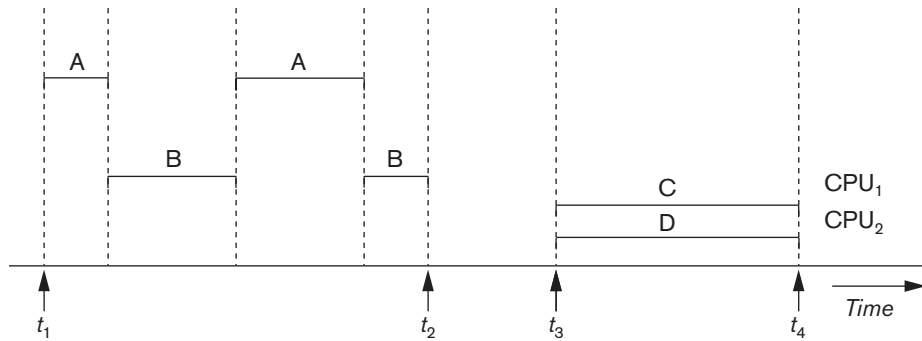
## 20.1 Introduction to Transaction Processing

In this section, we discuss the concepts of concurrent execution of transactions and recovery from transaction failures. Section 20.1.1 compares single-user and multiuser database systems and demonstrates how concurrent execution of transactions can take place in multiuser systems. Section 20.1.2 defines the concept of transaction and presents a simple model of transaction execution based on read and write database operations. This model is used as the basis for defining and formalizing concurrency control and recovery concepts. Section 20.1.3 uses informal examples to show why concurrency control techniques are needed in multiuser systems. Finally, Section 20.1.4 discusses why techniques are needed to handle recovery from system and transaction failures by discussing the different ways in which transactions can fail while executing.

### 20.1.1 Single-User versus Multiuser Systems

One criterion for classifying a database system is according to the number of users who can use the system **concurrently**. A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of users and travel agents concurrently. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems. In these systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the system.

Multiple users can access databases—and use computer systems—simultaneously because of the concept of **multiprogramming**, which allows the operating system of the computer to execute multiple programs—or **processes**—at the same time. A single



**Figure 20.1**  
Interleaved  
processing versus  
parallel processing  
of concurrent  
transactions.

central processing unit (CPU) can only execute at most one process at a time. However, **multiprogramming operating systems** execute some commands from one process, then suspend that process and execute some commands from the next process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved**, as illustrated in Figure 20.1, which shows two processes, A and B, executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 20.1. Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency**, so for the remainder of this chapter we assume this model. In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

### 20.1.2 Transactions, Database Items, Read and Write Operations, and DBMS Buffers

A **transaction** is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations—these can include insertion, deletion, modification (update), or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve

data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.

The *database model* that is used to present transaction processing concepts is simple when compared to the data models that we discussed earlier in the book, such as the relational model or the object model. A **database** is basically represented as a collection of *named data items*. The size of a data item is called its **granularity**. A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general. Each data item has a *unique name*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. If the item granularity is a single record, then the record id can be the item name. Using this simplified database model, the basic database access operations that a transaction can include are as follows:

- **read\_item(X)**. Reads a database item named *X* into a program variable. To simplify our notation, we assume that *the program variable is also named X*.
- **write\_item(X)**. Writes the value of program variable *X* into the database item named *X*.

As we discussed in Chapter 16, the basic unit of data transfer from disk to main memory is one disk page (disk block). Executing a `read_item(X)` command includes the following steps:

1. Find the address of the disk block that contains item *X*.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer). The size of the buffer is the same as the disk block size.
3. Copy item *X* from the buffer to the program variable named *X*.

Executing a `write_item(X)` command includes the following steps:

1. Find the address of the disk block that contains item *X*.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item *X* from the program variable named *X* into its correct location in the buffer.
4. Store the updated disk block from the buffer back to disk (either immediately or at some later point in time).

It is step 4 that actually updates the database on disk. Sometimes the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system. The DBMS will maintain in the **database cache**

a number of **data buffers** in main memory. Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed. When these buffers are all occupied, and additional database disk blocks must be copied into memory, some **buffer replacement policy** is used to choose which of the current occupied buffers is to be replaced. Some commonly used buffer replacement policies are **LRU** (least recently used). If the chosen buffer has been modified, it must be written back to disk before it is reused.<sup>1</sup> There are also buffer replacement policies that are specific to DBMS characteristics. We briefly discuss a few of these in Section 20.2.4.

A transaction includes `read_item` and `write_item` operations to access and update the database. Figure 20.2 shows examples of two very simple transactions. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that the transaction writes. For example, the read-set of  $T_1$  in Figure 20.2 is  $\{X, Y\}$  and its write-set is also  $\{X, Y\}$ .

Concurrency control and recovery mechanisms are mainly concerned with the database commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is *uncontrolled*, it may lead to problems, such as an inconsistent database. In the next section, we informally introduce some of the problems that may occur.

### 20.1.3 Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner. We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight. Each record includes the *number of reserved seats* on that flight as a *named (uniquely identifiable) data item*, among other information. Figure 20.2(a) shows a transaction  $T_1$  that *transfers*  $N$  reservations from one flight whose number of reserved seats is stored in the database item named  $X$  to another flight whose number of reserved seats is stored in the database item named  $Y$ . Figure 20.2(b) shows a simpler transaction  $T_2$  that just *reserves*  $M$  seats on the first flight ( $X$ ) referenced in transaction  $T_1$ .<sup>2</sup> To simplify our example, we do not show additional portions of the transactions, such as checking whether a flight has enough seats available before reserving additional seats.

When a database access program is written, it has the flight number, the flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute *many different transactions*, each with a different flight number, date, and number of seats to be booked. For concurrency control purposes, a transaction is a *particular execution* of a program on a specific date, flight, and number

---

<sup>1</sup>We will not discuss general-purpose buffer replacement policies here because they are typically discussed in operating systems texts.

<sup>2</sup>A similar, more commonly used example assumes a bank database, with one transaction doing a transfer of funds from account  $X$  to account  $Y$  and the other transaction doing a deposit to account  $X$ .



**Figure 20.2**

Two sample transactions.  
 (a) Transaction  $T_1$ .  
 (b) Transaction  $T_2$ .

(a)	<div data-bbox="571 167 745 202"> <math>T_1</math> </div> <div data-bbox="571 202 745 409"> <pre> read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y); </pre> </div>	(b)	<div data-bbox="878 167 1056 202"> <math>T_2</math> </div> <div data-bbox="878 202 1056 409"> <pre> read_item(X); X := X + M; write_item(X); </pre> </div>
-----	---	-----	--

of seats. In Figures 20.2(a) and (b), the transactions  $T_1$  and  $T_2$  are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items  $X$  and  $Y$  in the database. Next we discuss the types of problems we may encounter with these two simple transactions if they run concurrently.

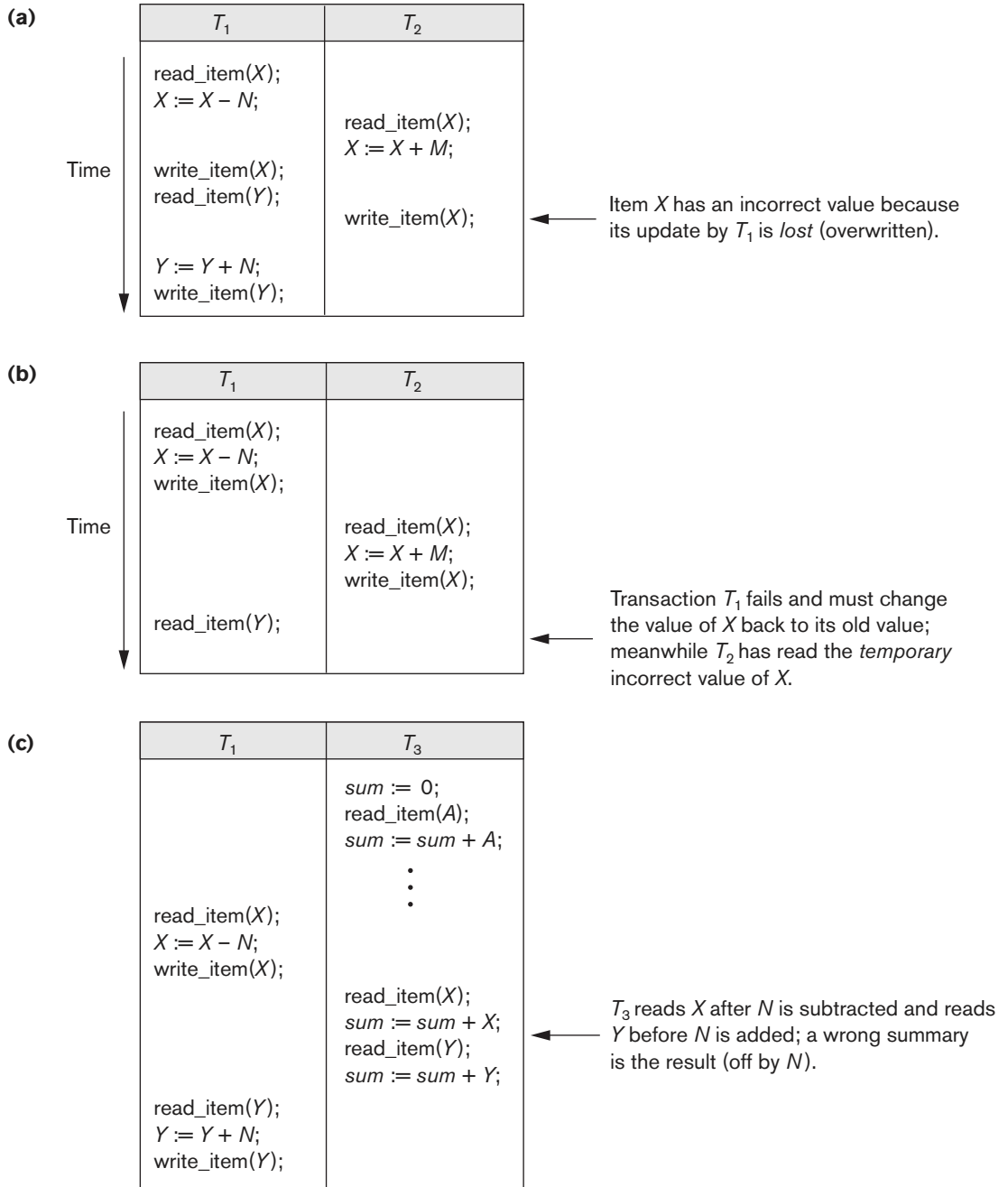
**The Lost Update Problem.** This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions  $T_1$  and  $T_2$  are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure 20.3(a); then the final value of item  $X$  is incorrect because  $T_2$  reads the value of  $X$  *before*  $T_1$  changes it in the database, and hence the updated value resulting from  $T_1$  is lost. For example, if  $X = 80$  at the start (originally there were 80 reservations on the flight),  $N = 5$  ( $T_1$  transfers 5 seat reservations from the flight corresponding to  $X$  to the flight corresponding to  $Y$ ), and  $M = 4$  ( $T_2$  reserves 4 seats on  $X$ ), the final result should be  $X = 79$ . However, in the interleaving of operations shown in Figure 20.3(a), it is  $X = 84$  because the update in  $T_1$  that removed the five seats from  $X$  was *lost*.

**The Temporary Update (or Dirty Read) Problem.** This problem occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 20.1.4). Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value. Figure 20.3(b) shows an example where  $T_1$  updates item  $X$  and then fails before completion, so the system must roll back  $X$  to its original value. Before it can do so, however, transaction  $T_2$  reads the *temporary* value of  $X$ , which will not be recorded permanently in the database because of the failure of  $T_1$ . The value of item  $X$  that is read by  $T_2$  is called *dirty data* because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem*.

**The Incorrect Summary Problem.** If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction  $T_3$  is calculating the total number of reservations on all the flights; meanwhile, transaction  $T_1$  is executing. If the interleaving of operations shown in Figure 20.3(c) occurs, the result of  $T_3$  will be off by an amount  $N$  because  $T_3$  reads the value of  $X$  *after*  $N$  seats have been subtracted from it but reads the value of  $Y$  *before* those  $N$  seats have been added to it.

**Figure 20.3**

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



**The Unrepeatable Read Problem.** Another problem that may occur is called *unrepeatable read*, where a transaction  $T$  reads the same item twice and the item is changed by another transaction  $T'$  between the two reads. Hence,  $T$  receives *different values* for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

#### 20.1.4 Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions. In the first case, the transaction is said to be **committed**, whereas in the second case, the transaction is **aborted**. The DBMS must not permit some operations of a transaction  $T$  to be applied to the database while other operations of  $T$  are not, because *the whole transaction* is a logical unit of database processing. If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

**Types of Failures.** Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. **A computer failure (system crash).** A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
2. **A transaction or system error.** Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error.<sup>3</sup> Additionally, the user may interrupt the transaction during its execution.
3. **Local errors or exception conditions detected by the transaction.** During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. An exception condition,<sup>4</sup> such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

---

<sup>3</sup>In general, a transaction should be thoroughly tested to ensure that it does not have any bugs (logical programming errors).

<sup>4</sup>Exception conditions, if programmed correctly, do not constitute transaction failures.

4. **Concurrency control enforcement.** The concurrency control method (see Chapter 21) may abort a transaction because it violates serializability (see Section 20.5), or it may abort one or more transactions to resolve a state of deadlock among several transactions (see Section 21.1.3). Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.
5. **Disk failure.** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. **Physical problems and catastrophes.** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task. We discuss recovery from failure in Chapter 22.

The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

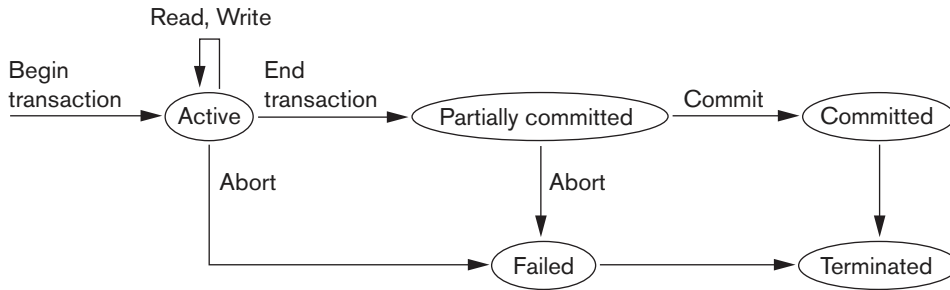
## 20.2 Transaction and System Concepts

In this section, we discuss additional concepts relevant to transaction processing. Section 20.2.1 describes the various states a transaction can be in and discusses other operations needed in transaction processing. Section 20.2.2 discusses the system log, which keeps information about transactions and data items that will be needed for recovery. Section 20.2.3 describes the concept of commit points of transactions and why they are important in transaction processing. Finally, Section 20.2.4 briefly discusses DBMS buffer replacement policies.

### 20.2.1 Transaction States and Additional Operations

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts (see Section 20.2.3). Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

- **BEGIN\_TRANSACTION.** This marks the beginning of transaction execution.
- **READ or WRITE.** These specify read or write operations on the database items that are executed as part of a transaction.
- **END\_TRANSACTION.** This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or

**Figure 20.4**

State transition diagram illustrating the states for transaction execution.

whether the transaction has to be aborted because it violates serializability (see Section 20.5) or for some other reason.

- **COMMIT\_TRANSACTION.** This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- **ROLLBACK (or ABORT).** This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be **undone**.

Figure 20.4 shows a state transition diagram that illustrates how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the **partially committed state**. At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not. Also, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log, discussed in the next section).<sup>5</sup> If these checks are successful, the transaction is said to have reached its commit point and enters the **committed state**. Commit points are discussed in more detail in Section 20.2.3. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

However, a transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The **terminated state** corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be *restarted* later—either automatically or after being resubmitted by the user—as brand new transactions.

<sup>5</sup>Optimistic concurrency control (see Section 21.4) also requires that certain checks are made at this point to ensure that the transaction did not interfere with other executing transactions.

### 20.2.2 The System Log

To be able to recover from failures that affect transactions, the system maintains a **log**<sup>6</sup> to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one (or more) main memory buffers, called the **log buffers**, hold the last part of the log file, so that log entries are first added to the log main memory buffer. When the **log buffer** is filled, or when certain other conditions occur, the log buffer is *appended to the end of the log file on disk*. In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures. The following are the types of entries—called **log records**—that are written to the log file and the corresponding action for each log record. In these entries, *T* refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:

1. [**start\_transaction**, *T*]. Indicates that transaction *T* has started execution.
2. [**write\_item**, *T*, *X*, *old\_value*, *new\_value*]. Indicates that transaction *T* has changed the value of database item *X* from *old\_value* to *new\_value*.
3. [**read\_item**, *T*, *X*]. Indicates that transaction *T* has read the value of database item *X*.
4. [**commit**, *T*]. Indicates that transaction *T* has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [**abort**, *T*]. Indicates that transaction *T* has been aborted.

Protocols for recovery that avoid cascading rollbacks (see Section 20.4.2)—which include nearly all practical protocols—*do not require* that READ operations are written to the system log. However, if the log is also used for other purposes—such as auditing (keeping track of all database operations)—then such entries can be included. Additionally, some recovery protocols require simpler WRITE entries that only include one of *new\_value* or *old\_value* instead of including both (see Section 20.4.2).

Notice that we are assuming that all permanent changes to the database occur within transactions, so the notion of recovery from a transaction failure amounts to either undoing or redoing transaction operations individually from the log. If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 22. Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to **undo** the effect of these WRITE operations of a transaction *T* by tracing backward through the log and resetting all items changed by a WRITE operation of *T* to their *old\_values*. **Redo** of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the sys-

---

<sup>6</sup>The log has sometimes been called the *DBMS journal*.

tem can be sure that all these *new\_values* have been written to the actual database on disk from the main memory buffers.<sup>7</sup>

### 20.2.3 Commit Point of a Transaction

A transaction *T* reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database have been recorded in the log. Beyond the commit point, the transaction is said to be **committed**, and its effect must be *permanently recorded* in the database. The transaction then writes a commit record [commit, *T*] into the log. If a system failure occurs, we can search back in the log for all transactions *T* that have written a [start\_transaction, *T*] record into the log but have not written their [commit, *T*] record yet; these transactions may have to be *rolled back* to *undo their effect* on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.

Notice that the log file must be kept on disk. As discussed in Chapter 16, updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk. As we mentioned earlier, it is common to keep one or more blocks of the log file in main memory buffers, called the **log buffer**, until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same log file buffer. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process if the contents of main memory are lost. Hence, *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called **force-writing** the log buffer to disk before committing a transaction.

### 20.2.4 DBMS-Specific Buffer Replacement Policies

The DBMS cache will hold the disk pages that contain information currently being processed in main memory buffers. If all the buffers in the DBMS cache are occupied and new disk pages are required to be loaded into main memory from disk, a **page replacement policy** is needed to select the particular buffers to be replaced. Some page replacement policies that have been developed specifically for database systems are briefly discussed next.

**Domain Separation (DS) Method.** In a DBMS, various types of disk pages exist: index pages, data file pages, log file pages, and so on. In this method, the DBMS cache is divided into separate domains (sets of buffers). Each domain handles one type of disk pages, and page replacements within each domain are han-

---

<sup>7</sup>Undo and redo are discussed more fully in Chapter 22.

dled via the basic LRU (least recently used) page replacement. Although this achieves better performance on average than basic LRU, it is a *static algorithm*, and so does not adapt to dynamically changing loads because the number of available buffers for each domain is predetermined. Several variations of the DS page replacement policy have been proposed, which add dynamic load-balancing features. For example, the **GRU** (Group LRU) gives each domain a priority level and selects pages from the lowest-priority level domain first for replacement, whereas another method dynamically changes the number of buffers in each domain based on current workload.

**Hot Set Method.** This page replacement algorithm is useful in queries that have to scan a set of pages repeatedly, such as when a join operation is performed using the nested-loop method (see Chapter 18). If the inner loop file is loaded completely into main memory buffers without replacement (the hot set), the join will be performed efficiently because each page in the outer loop file will have to scan all the records in the inner loop file to find join matches. The hot set method determines for each database processing algorithm the set of disk pages that will be accessed repeatedly, and it does not replace them until their processing is completed.

**The DBMIN Method.** This page replacement policy uses a model known as **QLSM** (query locality set model), which predetermines the pattern of page references for each algorithm for a particular type of database operation. We discussed various algorithms for relational operations such as **SELECT** and **JOIN** in Chapter 18. Depending on the type of access method, the file characteristics, and the algorithm used, the QLSM will estimate the number of main memory buffers needed for each file involved in the operation. The DBMIN page replacement policy will calculate a **locality set** using QLSM for each file instance involved in the query (some queries may reference the same file twice, so there would be a locality set for each file instance needed in the query). DBMIN then allocates the appropriate number of buffers to each file instance involved in the query based on the locality set for that file instance. The concept of locality set is analogous to the concept of *working set*, which is used in page replacement policies for processes by the operating system but there are multiple locality sets, one for each file instance in the query.

## 20.3 Desirable Properties of Transactions

Transactions should possess several properties, often called the **ACID** properties; they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

- **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
- **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.



- **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
- **Durability or permanency.** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The *atomicity property* requires that we execute a transaction to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs and of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS.<sup>8</sup> If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks (see Chapter 22) but does not eliminate all other problems.

The *durability property* is the responsibility of the *recovery subsystem* of the DBMS. In the next section, we introduce how recovery protocols enforce durability and atomicity and then discuss this in more detail in Chapter 22.

**Levels of Isolation.** There have been attempts to define the **level of isolation** of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to level 2 properties, repeatable reads.<sup>9</sup> Another type of isolation is called **snapshot isolation**, and several practical concurrency control methods are based on this. We shall discuss snapshot isolation in Section 20.6, and again in Chapter 21, Section 21.4.

<sup>8</sup>We will discuss concurrency control protocols in Chapter 21.

<sup>9</sup>The SQL syntax for isolation level discussed in Section 20.6 is closely related to these levels.

## 20.4 Characterizing Schedules Based on Recoverability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or **history**). In this section, first we define the concept of schedules, and then we characterize the types of schedules that facilitate recovery when failures occur. In Section 20.5, we characterize schedules in terms of the interference of participating transactions; this discussion leads to the concepts of serializability and serializable schedules.

### 20.4.1 Schedules (Histories) of Transactions

A **schedule** (or **history**)  $S$  of  $n$  transactions  $T_1, T_2, \dots, T_n$  is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule  $S$ . However, for each transaction  $T_i$  that participates in the schedule  $S$ , the operations of  $T_i$  in  $S$  must appear in the same order in which they occur in  $T_i$ . The order of operations in  $S$  is considered to be a *total ordering*, meaning that for any two operations in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders*, but we will assume for now total ordering of the operations in a schedule.

For the purpose of recovery and concurrency control, we are mainly interested in the `read_item` and `write_item` operations of the transactions, as well as the `commit` and `abort` operations. A shorthand notation for describing a schedule uses the symbols  $b$ ,  $r$ ,  $w$ ,  $e$ ,  $c$ , and  $a$  for the operations `begin_transaction`, `read_item`, `write_item`, `end_transaction`, `commit`, and `abort`, respectively, and appends as a *subscript* the transaction id (transaction number) to each operation in the schedule. In this notation, the database item  $X$  that is read or written follows the  $r$  and  $w$  operations in parentheses. In some schedules, we will only show the *read* and *write* operations, whereas in other schedules we will show additional operations, such as `commit` or `abort`. The schedule in Figure 20.3(a), which we shall call  $S_a$ , can be written as follows in this notation:

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Similarly, the schedule for Figure 20.3(b), which we call  $S_b$ , can be written as follows, if we assume that transaction  $T_1$  aborted after its `read_item(Y)` operation:

$$S_b: r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1;$$

**Conflicting Operations in a Schedule.** Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions: (1) they belong to *different transactions*; (2) they access the *same item*  $X$ ; and (3) *at least one* of the operations is a `write_item(X)`. For example, in schedule  $S_a$ , the operations  $r_1(X)$  and  $w_2(X)$  conflict, as do the operations  $r_2(X)$  and  $w_1(X)$ , and the operations  $w_1(X)$  and  $w_2(X)$ . However, the operations  $r_1(X)$  and  $r_2(X)$  do not conflict, since they are both read

operations; the operations  $w_2(X)$  and  $w_1(Y)$  do not conflict because they operate on distinct data items  $X$  and  $Y$ ; and the operations  $r_1(X)$  and  $w_1(X)$  do not conflict because they belong to the same transaction.

Intuitively, two operations are conflicting if changing their order can result in a different outcome. For example, if we change the order of the two operations  $r_1(X)$ ;  $w_2(X)$  to  $w_2(X)$ ;  $r_1(X)$ , then the value of  $X$  that is read by transaction  $T_1$  changes, because in the second ordering the value of  $X$  is read by  $r_1(X)$  *after* it is changed by  $w_2(X)$ , whereas in the first ordering the value is read *before* it is changed. This is called a **read-write conflict**. The other type is called a **write-write conflict** and is illustrated by the case where we change the order of two operations such as  $w_1(X)$ ;  $w_2(X)$  to  $w_2(X)$ ;  $w_1(X)$ . For a write-write conflict, the *last value* of  $X$  will differ because in one case it is written by  $T_2$  and in the other case by  $T_1$ . Notice that two read operations are not conflicting because changing their order makes no difference in outcome.

The rest of this section covers some theoretical definitions concerning schedules. A schedule  $S$  of  $n$  transactions  $T_1, T_2, \dots, T_n$  is said to be a **complete schedule** if the following conditions hold:

1. The operations in  $S$  are exactly those operations in  $T_1, T_2, \dots, T_n$ , including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction  $T_i$ , their relative order of appearance in  $S$  is the same as their order of appearance in  $T_i$ .
3. For any two conflicting operations, one of the two must occur before the other in the schedule.<sup>10</sup>

The preceding condition (3) allows for two *nonconflicting operations* to occur in the schedule without defining which occurs first, thus leading to the definition of a schedule as a **partial order** of the operations in the  $n$  transactions.<sup>11</sup> However, a total order must be specified in the schedule for any pair of conflicting operations (condition 3) and for any pair of operations from the same transaction (condition 2). Condition 1 simply states that all operations in the transactions must appear in the complete schedule. Since every transaction has either committed or aborted, a complete schedule will *not contain any active transactions* at the end of the schedule.

In general, it is difficult to encounter complete schedules in a transaction processing system because new transactions are continually being submitted to the system. Hence, it is useful to define the concept of the **committed projection**  $C(S)$  of a schedule  $S$ , which includes only the operations in  $S$  that belong to committed transactions—that is, transactions  $T_i$  whose commit operation  $c_i$  is in  $S$ .

<sup>10</sup>Theoretically, it is not necessary to determine an order between pairs of *nonconflicting* operations.

<sup>11</sup>In practice, most schedules have a total order of operations. If parallel processing is employed, it is theoretically possible to have schedules with partially ordered nonconflicting operations.

### 20.4.2 Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved. In some cases, it is even not possible to recover correctly after a failure. Hence, it is important to characterize the types of schedules for which *recovery is possible*, as well as those for which *recovery is relatively simple*. These characterizations do not actually provide the recovery algorithm; they only attempt to theoretically characterize the different types of schedules.

First, we would like to ensure that, once a transaction  $T$  is committed, it should *never* be necessary to roll back  $T$ . This ensures that the durability property of transactions is not violated (see Section 20.3). The schedules that theoretically meet this criterion are called *recoverable schedules*. A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable** and hence should not be permitted by the DBMS. The condition for a **recoverable schedule** is as follows: A schedule  $S$  is recoverable if no transaction  $T$  in  $S$  commits until all transactions  $T'$  that have written some item  $X$  that  $T$  reads have committed. A transaction  $T$  **reads** from transaction  $T'$  in a schedule  $S$  if some item  $X$  is first written by  $T'$  and later read by  $T$ . In addition,  $T'$  should not have been aborted before  $T$  reads item  $X$ , and there should be no transactions that write  $X$  after  $T'$  writes it and before  $T$  reads it (unless those transactions, if any, have aborted before  $T$  reads  $X$ ).

Some recoverable schedules may require a complex recovery process, as we shall see, but if sufficient information is kept (in the log), a recovery algorithm can be devised for any recoverable schedule. The (partial) schedules  $S_a$  and  $S_b$  from the preceding section are both recoverable, since they satisfy the above definition. Consider the schedule  $S_a'$  given below, which is the same as schedule  $S_a$  except that two commit operations have been added to  $S_a$ :

$S_a': r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$

$S_a'$  is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory (see Section 20.5). However, consider the two (partial) schedules  $S_c$  and  $S_d$  that follow:

$S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$   
 $S_d: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$   
 $S_e: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$

$S_c$  is not recoverable because  $T_2$  reads item  $X$  from  $T_1$ , but  $T_2$  commits before  $T_1$  commits. The problem occurs if  $T_1$  aborts after the  $c_2$  operation in  $S_c$ ; then the value of  $X$  that  $T_2$  read is no longer valid and  $T_2$  must be aborted *after* it is committed, leading to a schedule that is *not recoverable*. For the schedule to be recoverable, the  $c_2$  operation in  $S_c$  must be postponed until after  $T_1$  commits, as shown in  $S_d$ . If  $T_1$  aborts instead of committing, then  $T_2$  should also abort as shown in  $S_e$ , because the value of  $X$  it read is no longer valid. In  $S_e$ , aborting  $T_2$  is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule  $S_c$ .

In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of a committed transaction as durable is not violated. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur in some recoverable schedules, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule  $S_e$ , where transaction  $T_2$  has to be rolled back because it read item  $X$  from  $T_1$ , and  $T_1$  then aborted.

Because cascading rollback can be time-consuming—since numerous transactions can be rolled back (see Chapter 22)—it is important to characterize the schedules where this phenomenon is guaranteed not to occur. A schedule is said to be **cascadeless**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions. In this case, all items read will not be discarded because the transactions that wrote them have committed, so no cascading rollback will occur. To satisfy this criterion, the  $r_2(X)$  command in schedules  $S_d$  and  $S_e$  must be postponed until after  $T_1$  has committed (or aborted), thus delaying  $T_2$  but ensuring no cascading rollback if  $T_1$  aborts.

Finally, there is a third, more restrictive type of schedule, called a **strict schedule**, in which transactions can *neither read nor write* an item  $X$  until the last transaction that wrote  $X$  has committed (or aborted). Strict schedules simplify the recovery process. In a strict schedule, the process of undoing a `write_item(X)` operation of an aborted transaction is simply to restore the **before image** (old\_value or BFIM) of data item  $X$ . This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules. For example, consider schedule  $S_f$ :

$S_f: w_1(X, 5); w_2(X, 8); a_1;$

Suppose that the value of  $X$  was originally 9, which is the before image stored in the system log along with the  $w_1(X, 5)$  operation. If  $T_1$  aborts, as in  $S_f$ , the recovery procedure that restores the before image of an aborted write operation will restore the value of  $X$  to 9, even though it has already been changed to 8 by transaction  $T_2$ , thus leading to potentially incorrect results. Although schedule  $S_f$  is cascadeless, it is not a strict schedule, since it permits  $T_2$  to write item  $X$  even though the transaction  $T_1$  that last wrote  $X$  had not yet committed (or aborted). A strict schedule does not have this problem.

It is important to note that any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable. Suppose we have  $i$  transactions  $T_1, T_2, \dots, T_i$ , and their number of operations are  $n_1, n_2, \dots, n_i$ , respectively. If we make a set of *all possible schedules* of these transactions, we can divide the schedules into two disjoint subsets: recoverable and nonrecoverable. The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules. Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.

Most recovery protocols allow only strict schedules, so that the recovery process itself is not complicated (see Chapter 22).

## 20.5 Characterizing Schedules Based on Serializability

In the previous section, we characterized schedules based on their recoverability properties. Now we characterize the types of schedules that are always considered to be *correct* when concurrent transactions are executing. Such schedules are known as *serializable schedules*. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions  $T_1$  and  $T_2$  in Figure 20.2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

1. Execute all the operations of transaction  $T_1$  (in sequence) followed by all the operations of transaction  $T_2$  (in sequence).
2. Execute all the operations of transaction  $T_2$  (in sequence) followed by all the operations of transaction  $T_1$  (in sequence).

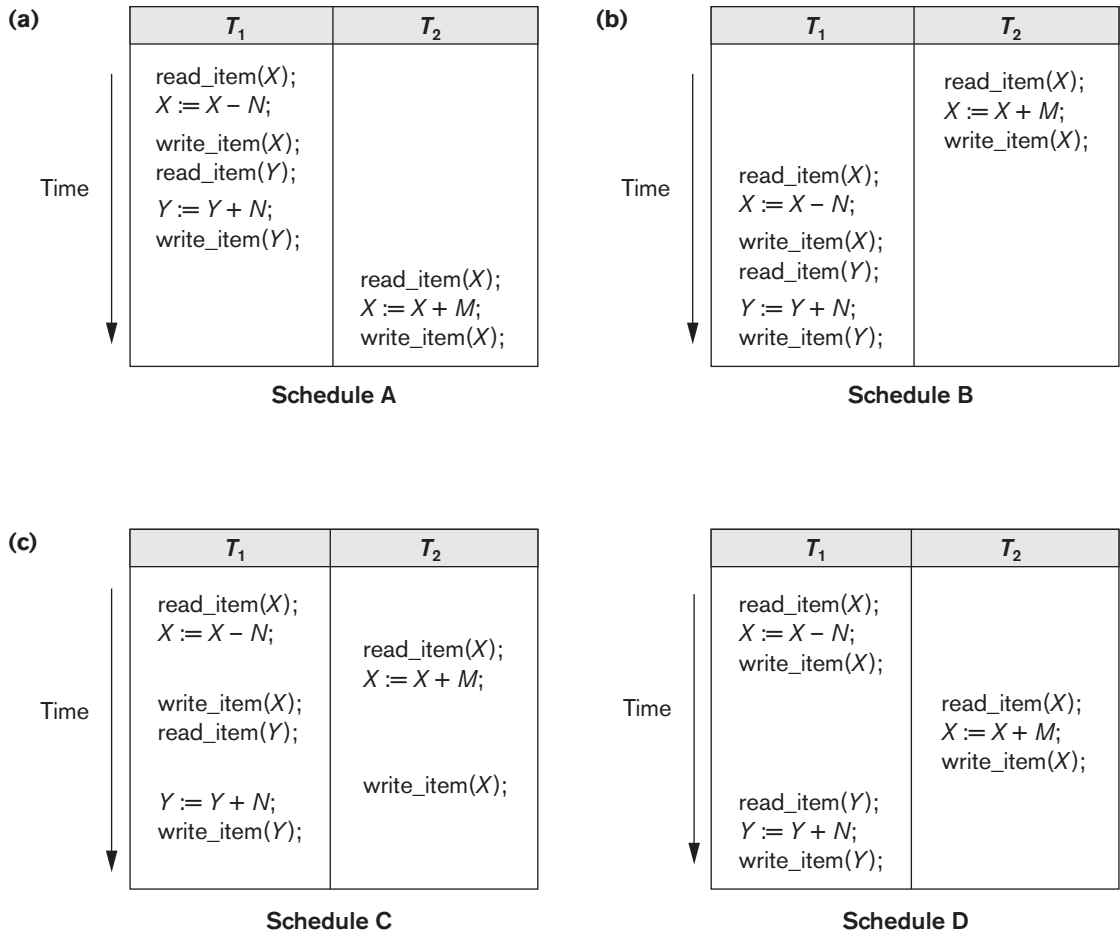
These two schedules—called *serial schedules*—are shown in Figures 20.5(a) and (b), respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Two possible schedules are shown in Figure 20.5(c). The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules. This section defines serializability and discusses how it may be used in practice.

### 20.5.1 Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figures 20.5(a) and (b) are called *serial* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order:  $T_1$  and then  $T_2$  in Figure 20.5(a), and  $T_2$  and then  $T_1$  in Figure 20.5(b). Schedules C and D in Figure 20.5(c) are called *nonserial* because each sequence interleaves operations from the two transactions.

Formally, a schedule  $S$  is **serial** if, for every transaction  $T$  participating in the schedule, all the operations of  $T$  are executed consecutively in the schedule; otherwise, the schedule is called **nonserial**. Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be *independent*, is that *every serial schedule is considered correct*. We can assume this because every transaction is assumed to be correct if executed on its own (according to the *consistency preservation* property of Section 20.3). Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation from the operations of other transactions, we get a correct end result.

The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O



**Figure 20.5**  
Examples of serial and nonserial schedules involving transactions  $T_1$  and  $T_2$ . (a) Serial schedule A:  $T_1$  followed by  $T_2$ . (b) Serial schedule B:  $T_2$  followed by  $T_1$ . (c) Two nonserial schedules C and D with interleaving of operations.

operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time. Additionally, if some transaction  $T$  is long, the other transactions must wait for  $T$  to complete all its operations before starting. Hence, serial schedules are *unacceptable* in practice. However, if we can determine which other schedules are *equivalent* to a serial schedule, we can allow these schedules to occur.

To illustrate our discussion, consider the schedules in Figure 20.5, and assume that the initial values of database items are  $X = 90$  and  $Y = 90$  and that  $N = 3$  and  $M = 2$ . After executing transactions  $T_1$  and  $T_2$ , we would expect the database values to be  $X = 89$  and  $Y = 93$ , according to the meaning of the transactions. Sure enough, executing either of the serial schedules A or B gives the correct results. Now consider



the nonserial schedules C and D. Schedule C (which is the same as Figure 20.3(a)) gives the results  $X = 92$  and  $Y = 93$ , in which the  $X$  value is erroneous, whereas schedule D gives the correct results.

Schedule C gives an erroneous result because of the *lost update problem* discussed in Section 20.1.3; transaction  $T_2$  reads the value of  $X$  before it is changed by transaction  $T_1$ , so only the effect of  $T_2$  on  $X$  is reflected in the database. The effect of  $T_1$  on  $X$  is *lost*, overwritten by  $T_2$ , leading to the incorrect result for item  $X$ . However, some nonserial schedules give the correct expected result, such as schedule D. We would like to determine which of the nonserial schedules *always* give a correct result and which may give erroneous results. The concept used to characterize schedules in this manner is that of serializability of a schedule.

The definition of *serializable schedule* is as follows: A schedule  $S$  of  $n$  transactions is **serializable** if it is *equivalent to some serial schedule* of the same  $n$  transactions. We will define the concept of *equivalence of schedules* shortly. Notice that there are  $n!$  possible serial schedules of  $n$  transactions and many more possible nonserial schedules. We can form two disjoint groups of the nonserial schedules—those that are equivalent to one (or more) of the serial schedules and hence are serializable, and those that are not equivalent to *any* serial schedule and hence are not serializable.

Saying that a nonserial schedule  $S$  is serializable is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct. The remaining question is: When are two schedules considered *equivalent*?

There are several ways to define schedule equivalence. The simplest but least satisfactory definition involves comparing the effects of the schedules on the database. Two schedules are called **result equivalent** if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state. For example, in Figure 20.6, schedules  $S_1$  and  $S_2$  will produce the same final database state if they execute on a database with an initial value of  $X = 100$ ; however, for other initial values of  $X$ , the schedules are *not* result equivalent. Additionally, these schedules execute different transactions, so they definitely should not be considered equivalent. Hence, result equivalence alone cannot be used to define equivalence of schedules. The safest and most general approach to defining schedule equivalence is to focus only on the *read\_item* and *write\_item* operations of the transactions, and not make any assumptions about the other internal operations included in the transactions. For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules *in the same order*. Two definitions of equivalence of schedules are generally used: *conflict equivalence* and *view equivalence*. We discuss conflict equivalence next, which is the more commonly used definition.

**Conflict Equivalence of Two Schedules.** Two schedules are said to be **conflict equivalent** if the relative order of any two *conflicting operations* is the same in both schedules. Recall from Section 20.4.1 that two operations in a schedule are said to



**Figure 20.6**  
Two schedules that are result equivalent for the initial value of  $X = 100$  but are not result equivalent in general.

$S_1$	$S_2$
<code>read_item(X);</code> <code>X := X + 10;</code> <code>write_item(X);</code>	<code>read_item(X);</code> <code>X := X * 1.1;</code> <code>write_item(X);</code>

*conflict* if they belong to different transactions, access the same database item, and either both are `write_item` operations or one is a `write_item` and the other a `read_item`. If two conflicting operations are applied in *different orders* in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent. For example, as we discussed in Section 20.4.1, if a read and write operation occur in the order  $r_1(X), w_2(X)$  in schedule  $S_1$ , and in the reverse order  $w_2(X), r_1(X)$  in schedule  $S_2$ , the value read by  $r_1(X)$  can be different in the two schedules. Similarly, if two write operations occur in the order  $w_1(X), w_2(X)$  in  $S_1$ , and in the reverse order  $w_2(X), w_1(X)$  in  $S_2$ , the next  $r(X)$  operation in the two schedules will read potentially different values; or if these are the last operations writing item  $X$  in the schedules, the final value of item  $X$  in the database will be different.

**Serializable Schedules.** Using the notion of conflict equivalence, we define a schedule  $S$  to be **serializable**<sup>12</sup> if it is (conflict) equivalent to some serial schedule  $S'$ . In such a case, we can reorder the *nonconflicting* operations in  $S$  until we form the equivalent serial schedule  $S'$ . According to this definition, schedule D in Figure 20.5(c) is equivalent to the serial schedule A in Figure 20.5(a). In both schedules, the `read_item(X)` of  $T_2$  reads the value of  $X$  written by  $T_1$ , whereas the other `read_item` operations read the database values from the initial database state. Additionally,  $T_1$  is the last transaction to write  $Y$ , and  $T_2$  is the last transaction to write  $X$  in both schedules. Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule. Notice that the operations  $r_1(Y)$  and  $w_1(Y)$  of schedule D do not conflict with the operations  $r_2(X)$  and  $w_2(X)$ , since they access different data items. Therefore, we can move  $r_1(Y), w_1(Y)$  before  $r_2(X), w_2(X)$ , leading to the equivalent serial schedule  $T_1, T_2$ .

Schedule C in Figure 20.5(c) is not equivalent to either of the two possible serial schedules A and B, and hence is *not serializable*. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails because  $r_2(X)$  and  $w_1(X)$  conflict, which means that we cannot move  $r_2(X)$  down to get the equivalent serial schedule  $T_1, T_2$ . Similarly, because  $w_1(X)$  and  $w_2(X)$  conflict, we cannot move  $w_1(X)$  down to get the equivalent serial schedule  $T_2, T_1$ .

Another, more complex definition of equivalence—called *view equivalence*, which leads to the concept of view serializability—is discussed in Section 20.5.4.

<sup>12</sup>We will use *serializable* to mean conflict serializable. Another definition of serializable used in practice (see Section 20.6) is to have repeatable reads, no dirty reads, and no phantom records (see Section 22.7.1 for a discussion on phantoms).

## 20.5.2 Testing for Serializability of a Schedule

There is a simple algorithm for determining whether a particular schedule is (conflict) serializable or not. Most concurrency control methods do *not* actually test for serializability. Rather protocols, or rules, are developed that guarantee that any schedule that follows these rules will be serializable. Some methods guarantee serializability in most cases, but do not guarantee it absolutely, in order to reduce the overhead of concurrency control. We discuss the algorithm for testing conflict serializability of schedules here to gain a better understanding of these concurrency control protocols, which are discussed in Chapter 21.

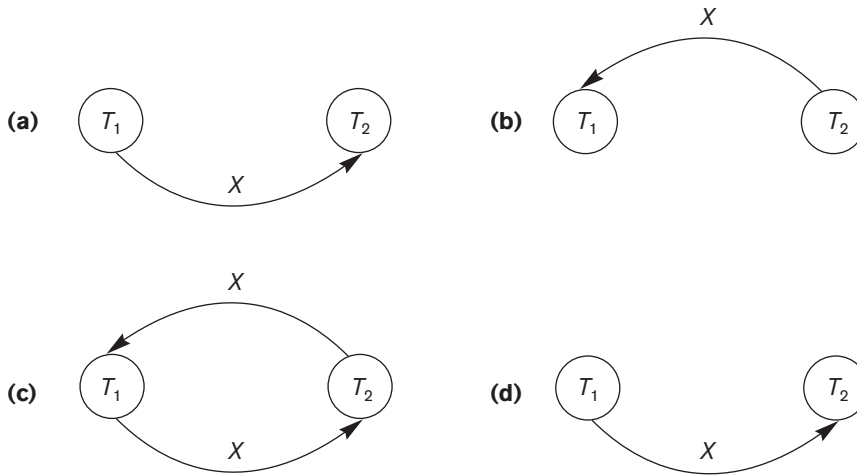
Algorithm 20.1 can be used to test a schedule for conflict serializability. The algorithm looks at only the `read_item` and `write_item` operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a **directed graph**  $G = (N, E)$  that consists of a set of nodes  $N = \{T_1, T_2, \dots, T_n\}$  and a set of directed edges  $E = \{e_1, e_2, \dots, e_m\}$ . There is one node in the graph for each transaction  $T_i$  in the schedule. Each edge  $e_i$  in the graph is of the form  $(T_j \rightarrow T_k)$ ,  $1 \leq j \leq n$ ,  $1 \leq k \leq n$ , where  $T_j$  is the **starting node** of  $e_i$  and  $T_k$  is the **ending node** of  $e_i$ . Such an edge from node  $T_j$  to node  $T_k$  is created by the algorithm if a pair of conflicting operations exist in  $T_j$  and  $T_k$  and the conflicting operation in  $T_j$  appears in the schedule *before* the *conflicting operation* in  $T_k$ .

### Algorithm 20.1. Testing Conflict Serializability of a Schedule S

1. For each transaction  $T_i$  participating in schedule S, create a node labeled  $T_i$  in the precedence graph.
2. For each case in S where  $T_j$  executes a `read_item(X)` after  $T_i$  executes a `write_item(X)`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
3. For each case in S where  $T_j$  executes a `write_item(X)` after  $T_i$  executes a `read_item(X)`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
4. For each case in S where  $T_j$  executes a `write_item(X)` after  $T_i$  executes a `write_item(X)`, create an edge  $(T_i \rightarrow T_j)$  in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

The precedence graph is constructed as described in Algorithm 20.1. If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable. A **cycle** in a directed graph is a **sequence of edges**  $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), \dots, (T_i \rightarrow T_j))$  with the property that the starting node of each edge—except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge (the sequence starts and ends at the same node).

In the precedence graph, an edge from  $T_i$  to  $T_j$  means that transaction  $T_i$  must come before transaction  $T_j$  in any serial schedule that is equivalent to S, because two conflicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, we can create an **equivalent serial schedule**  $S'$  that is equivalent to S, by ordering the transactions that participate in S as follows: Whenever an edge exists

**Figure 20.7**

Constructing the precedence graphs for schedules A to D from Figure 20.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

in the precedence graph from  $T_i$  to  $T_j$ ,  $T_i$  must appear before  $T_j$  in the equivalent serial schedule  $S'$ .<sup>13</sup> Notice that the edges ( $T_i \rightarrow T_j$ ) in a precedence graph can optionally be labeled by the name(s) of the data item(s) that led to creating the edge. Figure 20.7 shows such labels on the edges. When checking for a cycle, the labels are not relevant.

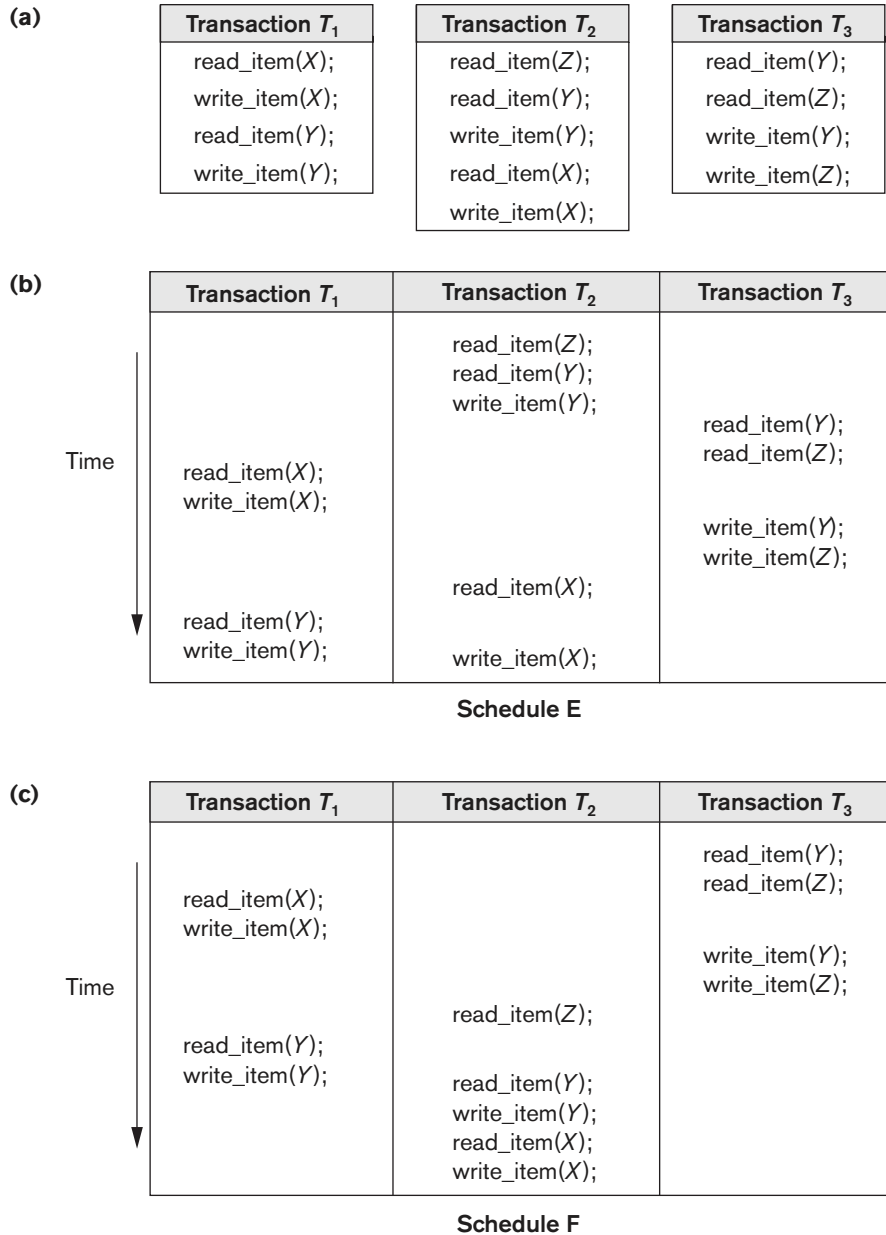
In general, several serial schedules can be equivalent to  $S$  if the precedence graph for  $S$  has no cycle. However, if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so  $S$  is not serializable. The precedence graphs created for schedules A to D, respectively, in Figure 20.5 appear in Figures 20.7(a) to (d). The graph for schedule C has a cycle, so it is not serializable. The graph for schedule D has no cycle, so it is serializable, and the equivalent serial schedule is  $T_1$  followed by  $T_2$ . The graphs for schedules A and B have no cycles, as expected, because the schedules are serial and hence serializable.

Another example, in which three transactions participate, is shown in Figure 20.8. Figure 20.8(a) shows the read\_item and write\_item operations in each transaction. Two schedules  $E$  and  $F$  for these transactions are shown in Figures 20.8(b) and (c), respectively, and the precedence graphs for schedules  $E$  and  $F$  are shown in Figures 20.8(d) and (e). Schedule  $E$  is not serializable because the corresponding precedence graph has cycles. Schedule  $F$  is serializable, and the serial schedule equivalent to  $F$  is shown in Figure 20.8(e). Although only one equivalent serial schedule exists for  $F$ , in general there may be more than one equivalent serial schedule for a serializable schedule. Figure 20.8(f) shows a precedence graph representing a schedule

<sup>13</sup>This process of ordering the nodes of an acyclic graph is known as *topological sorting*.

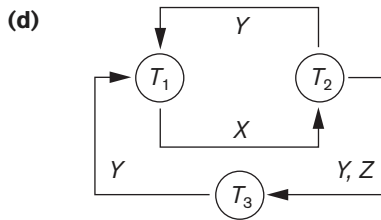
**Figure 20.8**

Another example of serializability testing. (a) The read and write operations of three transactions  $T_1$ ,  $T_2$ , and  $T_3$ . (b) Schedule E. (c) Schedule F.

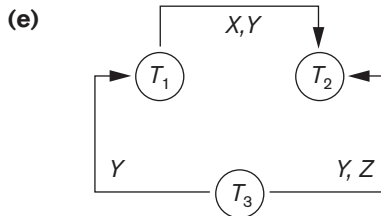
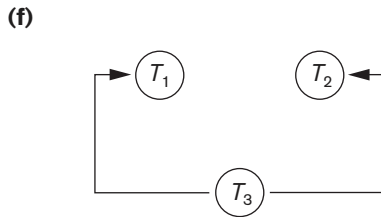


**Figure 20.8 (continued)**

Another example of serializability testing. (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) Precedence graph with two equivalent serial schedules.

**Equivalent serial schedules**

None

**Reason**Cycle  $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$ Cycle  $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$ **Equivalent serial schedules** $T_3 \rightarrow T_1 \rightarrow T_2$ **Equivalent serial schedules** $T_3 \rightarrow T_1 \rightarrow T_2$  $T_3 \rightarrow T_2 \rightarrow T_1$ 

that has two equivalent serial schedules. To find an equivalent serial schedule, start with a node that does not have any incoming edges, and then make sure that the node order for every edge is not violated.

### 20.5.3 How Serializability Is Used for Concurrency Control

As we discussed earlier, saying that a schedule  $S$  is (conflict) serializable—that is,  $S$  is (conflict) equivalent to a serial schedule—is tantamount to saying that  $S$  is correct. Being *serializable* is distinct from being *serial*, however. A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for a long transaction to delay other transactions, thus slowing down transaction processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness. In practice, it is difficult to test for the serializability of a schedule. The interleaving of operations from concurrent transactions—which are usually executed as processes by the operating system—is typically determined by the operating system scheduler, which allocates

resources to all processes. Factors such as system load, time of transaction submission, and priorities of processes contribute to the ordering of operations in a schedule. Hence, it is difficult to determine how the operations of a schedule will be interleaved beforehand to ensure serializability.

If transactions are executed at will and then the resulting schedule is tested for serializability, we must cancel the effect of the schedule if it turns out not to be serializable. This is a serious problem that makes this approach impractical. The approach taken in most commercial DBMSs is to design **protocols** (sets of rules) that—if followed by *every* individual transaction or if enforced by a DBMS concurrency control subsystem—will ensure serializability of *all schedules in which the transactions participate*. Some protocols may allow nonserializable schedules in rare cases to reduce the overhead of the concurrency control method (see Section 20.6).

Another problem is that transactions are submitted continuously to the system, so it is difficult to determine when a schedule begins and when it ends. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule  $S$ . Recall from Section 20.4.1 that the *committed projection*  $C(S)$  of a schedule  $S$  includes only the operations in  $S$  that belong to committed transactions. We can theoretically define a schedule  $S$  to be serializable if its committed projection  $C(S)$  is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.

In Chapter 21, we discuss a number of different concurrency control protocols that guarantee serializability. The most common technique, called *two-phase locking*, is based on locking data items to prevent concurrent transactions from interfering with one another, and enforcing an additional condition that guarantees serializability. This is used in some commercial DBMSs. We will also discuss a protocol based on the concept of *snapshot isolation* that ensures serializability in most but not all cases; this is used in some commercial DBMSs because it has less overhead than the two-phase locking protocol. Other protocols have been proposed<sup>14</sup>; these include *timestamp ordering*, where each transaction is assigned a unique timestamp and the protocol ensures that any conflicting operations are executed in the order of the transaction timestamps; *multiversion protocols*, which are based on maintaining multiple versions of data items; and *optimistic* (also called *certification* or *validation*) *protocols*, which check for possible serializability violations after the transactions terminate but before they are permitted to commit.

### 20.5.4 View Equivalence and View Serializability

In Section 20.5.1, we defined the concepts of conflict equivalence of schedules and conflict serializability. Another less restrictive definition of equivalence of schedules is called *view equivalence*. This leads to another definition of serializability

---

<sup>14</sup>These other protocols have not been incorporated much into commercial systems; most relational DBMSs use some variation of two-phase locking or snapshot isolation.

called *view serializability*. Two schedules  $S$  and  $S'$  are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in  $S$  and  $S'$ , and  $S$  and  $S'$  include the same operations of those transactions.
2. For any operation  $r_i(X)$  of  $T_i$  in  $S$ , if the value of  $X$  read by the operation has been written by an operation  $w_j(X)$  of  $T_j$  (or if it is the original value of  $X$  before the schedule started), the same condition must hold for the value of  $X$  read by operation  $r_i(X)$  of  $T_i$  in  $S'$ .
3. If the operation  $w_k(Y)$  of  $T_k$  is the last operation to write item  $Y$  in  $S$ , then  $w_k(Y)$  of  $T_k$  must also be the last operation to write item  $Y$  in  $S'$ .

The idea behind view equivalence is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to *see the same view* in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules. A schedule  $S$  is said to be **view serializable** if it is view equivalent to a serial schedule.

The definitions of conflict serializability and view serializability are similar if a condition known as the **constrained write assumption** (or **no blind writes**) holds on all transactions in the schedule. This condition states that any write operation  $w_i(X)$  in  $T_i$  is preceded by a  $r_i(X)$  in  $T_i$  and that the value written by  $w_i(X)$  in  $T_i$  depends only on the value of  $X$  read by  $r_i(X)$ . This assumes that computation of the new value of  $X$  is a function  $f(X)$  based on the old value of  $X$  read from the database. A **blind write** is a write operation in a transaction  $T$  on an item  $X$  that is not dependent on the old value of  $X$ , so it is not preceded by a read of  $X$  in the transaction  $T$ .

The definition of view serializability is less restrictive than that of conflict serializability under the **unconstrained write assumption**, where the value written by an operation  $w_i(X)$  in  $T_i$  can be independent of its old value. This is possible when *blind writes* are allowed, and it is illustrated by the following schedule  $S_g$  of three transactions  $T_1$ :  $r_1(X)$ ;  $w_1(X)$ ;  $T_2$ :  $w_2(X)$ ; and  $T_3$ :  $w_3(X)$ :

$S_g$ :  $r_1(X)$ ;  $w_2(X)$ ;  $w_1(X)$ ;  $w_3(X)$ ;  $c_1$ ;  $c_2$ ;  $c_3$ ;

In  $S_g$  the operations  $w_2(X)$  and  $w_3(X)$  are blind writes, since  $T_2$  and  $T_3$  do not read the value of  $X$ . The schedule  $S_g$  is view serializable, since it is view equivalent to the serial schedule  $T_1, T_2, T_3$ . However,  $S_g$  is not conflict serializable, since it is not conflict equivalent to any serial schedule (as an exercise, the reader should construct the serializability graph for  $S_g$  and check for cycles). It has been shown that any conflict-serializable schedule is also view serializable but not vice versa, as illustrated by the preceding example. There is an algorithm to test whether a schedule  $S$  is view serializable or not. However, the problem of testing for view serializability has been shown to be NP-hard, meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely.

### 20.5.5 Other Types of Equivalence of Schedules

Serializability of schedules is sometimes considered to be too restrictive as a condition for ensuring the correctness of concurrent executions. Some applications can produce schedules that are correct by satisfying conditions less stringent than either conflict serializability or view serializability. An example is the type of transactions known as **debit-credit transactions**—for example, those that apply deposits and withdrawals to a data item whose value is the current balance of a bank account. The semantics of debit-credit operations is that they update the value of a data item  $X$  by either subtracting from or adding to the value of the data item. Because addition and subtraction operations are commutative—that is, they can be applied in any order—it is possible to produce correct schedules that are not serializable. For example, consider the following transactions, each of which may be used to transfer an amount of money between two bank accounts:

$$\begin{aligned} T_1: & r_1(X); X := \{equal\} X - 10; w_1(X); r_1(Y); Y := \{equal\} Y + 10; w_1(Y); \\ T_2: & r_2(Y); Y := \{equal\} Y - 20; w_2(Y); r_2(X); X := \{equal\} X + 20; w_2(X); \end{aligned}$$

Consider the following nonserializable schedule  $S_h$  for the two transactions:

$$S_h: r_1(X); w_1(X); r_2(Y); w_2(Y); r_1(Y); w_1(Y); r_2(X); w_2(X);$$

With the additional knowledge, or **semantics**, that the operations between each  $r_i(I)$  and  $w_i(I)$  are commutative, we know that the order of executing the sequences consisting of (read, update, write) is not important as long as each (read, update, write) sequence by a particular transaction  $T_i$  on a particular item  $I$  is not interrupted by conflicting operations. Hence, the schedule  $S_h$  is considered to be correct even though it is not serializable. Researchers have been working on extending concurrency control theory to deal with cases where serializability is considered to be too restrictive as a condition for correctness of schedules. Also, in certain domains of applications, such as computer-aided design (CAD) of complex systems like aircraft, design transactions last over a long time period. In such applications, more relaxed schemes of concurrency control have been proposed to maintain consistency of the database, such as *eventual consistency*. We shall discuss eventual consistency in the context of distributed databases in Chapter 23.

## 20.6 Transaction Support in SQL

In this section, we give a brief introduction to transaction support in SQL. There are many more details, and the newer standards have more commands for transaction processing. The basic definition of an SQL transaction is similar to our already defined concept of a transaction. That is, it is a logical unit of work and is guaranteed to be atomic. A single SQL statement is always considered to be atomic—either it completes execution without an error or it fails and leaves the database unchanged.



With SQL, there is no explicit `Begin_Transaction` statement. Transaction initiation is done implicitly when particular SQL statements are encountered. However, every transaction must have an explicit end statement, which is either a `COMMIT` or a `ROLLBACK`. Every transaction has certain characteristics attributed to it. These characteristics are specified by a `SET TRANSACTION` statement in SQL. The characteristics are the *access mode*, the *diagnostic area size*, and the *isolation level*.

The **access mode** can be specified as `READ ONLY` or `READ WRITE`. The default is `READ WRITE`, unless the isolation level of `READ UNCOMMITTED` is specified (see below), in which case `READ ONLY` is assumed. A mode of `READ WRITE` allows select, update, insert, delete, and create commands to be executed. A mode of `READ ONLY`, as the name implies, is simply for data retrieval.

The **diagnostic area size** option, `DIAGNOSTIC SIZE  $n$` , specifies an integer value  $n$ , which indicates the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information (errors or exceptions) to the user or program on the  $n$  most recently executed SQL statement.

The **isolation level** option is specified using the statement `ISOLATION LEVEL <isolation>`, where the value for <isolation> can be `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, or `SERIALIZABLE`.<sup>15</sup> The default isolation level is `SERIALIZABLE`, although some systems use `READ COMMITTED` as their default. The use of the term `SERIALIZABLE` here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms,<sup>16</sup> and it is thus not identical to the way serializability was defined earlier in Section 20.5. If a transaction executes at a lower isolation level than `SERIALIZABLE`, then one or more of the following three violations may occur:

1. **Dirty read.** A transaction  $T_1$  may read the update of a transaction  $T_2$ , which has not yet committed. If  $T_2$  fails and is aborted, then  $T_1$  would have read a value that does not exist and is incorrect.
2. **Nonrepeatable read.** A transaction  $T_1$  may read a given value from a table. If another transaction  $T_2$  later updates that value and  $T_1$  reads that value again,  $T_1$  will see a different value.
3. **Phantoms.** A transaction  $T_1$  may read a set of rows from a table, perhaps based on some condition specified in the SQL `WHERE`-clause. Now suppose that a transaction  $T_2$  inserts a new row  $r$  that also satisfies the `WHERE`-clause condition used in  $T_1$ , into the table used by  $T_1$ . The record  $r$  is called a **phantom record** because it was not there when  $T_1$  starts but is there when  $T_1$  ends.  $T_1$  may or may not see the phantom, a row that previously did not exist. If the equivalent serial order is  $T_1$  followed by  $T_2$ , then the record  $r$  should not be seen; but if it is  $T_2$  followed by  $T_1$ , then the phantom record should be in the result given to  $T_1$ . If the system cannot ensure the correct behavior, then it does not deal with the phantom record problem.

<sup>15</sup>These are similar to the *isolation levels* discussed briefly at the end of Section 20.3.

<sup>16</sup>The dirty read and unrepeatable read problems were discussed in Section 20.1.3. Phantoms are discussed in Section 22.7.1.

**Table 20.1** Possible Violations Based on Isolation Levels as Defined in SQL

Isolation Level	Type of Violation		
	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Table 20.1 summarizes the possible violations for the different isolation levels. An entry of *Yes* indicates that a violation is possible and an entry of *No* indicates that it is not possible. READ UNCOMMITTED is the most forgiving, and SERIALIZABLE is the most restrictive in that it avoids all three of the problems mentioned above.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

The above transaction consists of first inserting a new row in the EMPLOYEE table and then updating the salary of all employees who work in department 2. If an error occurs on any of the SQL statements, the entire transaction is rolled back. This implies that any updated salary (by this transaction) would be restored to its previous value and that the newly inserted row would be removed.

As we have seen, SQL provides a number of transaction-oriented features. The DBA or database programmers can take advantage of these options to try improving transaction performance by relaxing serializability if that is acceptable for their applications.

**Snapshot Isolation.** Another isolation level, known as snapshot isolation, is used in some commercial DBMSs, and some concurrency control protocols exist that are based on this concept. The basic definition of **snapshot isolation** is that a transaction sees the data items that it reads based on the committed values of the items in the *database snapshot* (or database state) when the transaction starts. Snapshot isolation will ensure that the phantom record problem does not occur, since

the database transaction, or in some cases the database statement, will only see the records that were committed in the database at the time the transaction starts. Any insertions, deletions, or updates that occur after the transaction starts will not be seen by the transaction. We will discuss a concurrency control protocol based on this concept in Chapter 21.

## 20.7 Summary

In this chapter, we discussed DBMS concepts for transaction processing. We introduced the concept of a database transaction and the operations relevant to transaction processing in Section 20.1. We compared single-user systems to multiuser systems and then presented examples of how uncontrolled execution of concurrent transactions in a multiuser system can lead to incorrect results and database values in Section 20.1.1. We also discussed the various types of failures that may occur during transaction execution in Section 20.1.4.

Next, in Section 20.2, we introduced the typical states that a transaction passes through during execution, and discussed several concepts that are used in recovery and concurrency control methods. The system log (Section 20.2.2) keeps track of database accesses, and the system uses this information to recover from failures. A transaction can succeed and reach its commit point, or it can fail and has to be rolled back. A committed transaction (Section 20.2.3) has its changes permanently recorded in the database. In Section 20.3, we presented an overview of the desirable properties of transactions—atomicity, consistency preservation, isolation, and durability—which are often referred to as the ACID properties.

Then we defined a schedule (or history) as an execution sequence of the operations of several transactions with interleaving in Section 20.4.1. We characterized schedules in terms of their recoverability in Section 20.4.2. Recoverable schedules ensure that, once a transaction commits, it never needs to be undone. Cascadeless schedules add an additional condition to ensure that no aborted transaction requires the cascading abort of other transactions. Strict schedules provide an even stronger condition that allows a simple recovery scheme consisting of restoring the old values of items that have been changed by an aborted transaction.

Then in Section 20.5 we defined the equivalence of schedules and saw that a serializable schedule is equivalent to some serial schedule. We defined the concepts of conflict equivalence and view equivalence. A serializable schedule is considered correct. We presented an algorithm for testing the (conflict) serializability of a schedule in Section 20.5.2. We discussed why testing for serializability is impractical in a real system, although it can be used to define and verify concurrency control protocols in Section 20.5.3, and we briefly mentioned less restrictive definitions of schedule equivalence in Sections 20.5.4 and 20.5.5. Finally, in Section 20.6, we gave a brief overview of how transaction concepts are used in practice within SQL, and we introduced the concept of snapshot isolation, which is used in several commercial DBMSs.

## Review Questions

- 20.1. What is meant by the concurrent execution of database transactions in a multiuser system? Discuss why concurrency control is needed, and give informal examples.
- 20.2. Discuss the different types of failures. What is meant by catastrophic failure?
- 20.3. Discuss the actions taken by the `read_item` and `write_item` operations on a database.
- 20.4. Draw a state diagram and discuss the typical states that a transaction goes through during execution.
- 20.5. What is the system log used for? What are the typical kinds of records in a system log? What are transaction commit points, and why are they important?
- 20.6. Discuss the atomicity, durability, isolation, and consistency preservation properties of a database transaction.
- 20.7. What is a schedule (history)? Define the concepts of recoverable, cascade-less, and strict schedules, and compare them in terms of their recoverability.
- 20.8. Discuss the different measures of transaction equivalence. What is the difference between conflict equivalence and view equivalence?
- 20.9. What is a serial schedule? What is a serializable schedule? Why is a serial schedule considered correct? Why is a serializable schedule considered correct?
- 20.10. What is the difference between the constrained write and the unconstrained write assumptions? Which is more realistic?
- 20.11. Discuss how serializability is used to enforce concurrency control in a database system. Why is serializability sometimes considered too restrictive as a measure of correctness for schedules?
- 20.12. Describe the four levels of isolation in SQL. Also discuss the concept of snapshot isolation and its effect on the phantom record problem.
- 20.13. Define the violations caused by each of the following: dirty read, nonrepeatable read, and phantoms.

## Exercises

- 20.14. Change transaction  $T_2$  in Figure 20.2(b) to read

```

read_item(X);
X := X + M;
if X > 90 then exit
else write_item(X);

```

Discuss the final result of the different schedules in Figures 20.3(a) and (b), where  $M = 2$  and  $N = 2$ , with respect to the following questions: Does adding the above condition change the final outcome? Does the outcome obey the implied consistency rule (that the capacity of  $X$  is 90)?

- 20.15.** Repeat Exercise 20.14, adding a check in  $T_1$  so that  $Y$  does not exceed 90.
- 20.16.** Add the operation commit at the end of each of the transactions  $T_1$  and  $T_2$  in Figure 20.2, and then list all possible schedules for the modified transactions. Determine which of the schedules are recoverable, which are cascade-less, and which are strict.
- 20.17.** List all possible schedules for transactions  $T_1$  and  $T_2$  in Figure 20.2, and determine which are conflict serializable (correct) and which are not.
- 20.18.** How many *serial* schedules exist for the three transactions in Figure 20.8(a)? What are they? What is the total number of possible schedules?
- 20.19.** Write a program to create all possible schedules for the three transactions in Figure 20.8(a), and to determine which of those schedules are conflict serializable and which are not. For each conflict-serializable schedule, your program should print the schedule and list all equivalent serial schedules.
- 20.20.** Why is an explicit transaction end statement needed in SQL but not an explicit begin statement?
- 20.21.** Describe situations where each of the different isolation levels would be useful for transaction processing.
- 20.22.** Which of the following schedules is (conflict) serializable? For each serializable schedule, determine the equivalent serial schedules.
- $r_1(X); r_3(X); w_1(X); r_2(X); w_3(X);$
  - $r_1(X); r_3(X); w_3(X); w_1(X); r_2(X);$
  - $r_3(X); r_2(X); w_3(X); r_1(X); w_1(X);$
  - $r_3(X); r_2(X); r_1(X); w_3(X); w_1(X);$
- 20.23.** Consider the three transactions  $T_1$ ,  $T_2$ , and  $T_3$ , and the schedules  $S_1$  and  $S_2$  given below. Draw the serializability (precedence) graphs for  $S_1$  and  $S_2$ , and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).
- $T_1: r_1(X); r_1(Z); w_1(X);$   
 $T_2: r_2(Z); r_2(Y); w_2(Z); w_2(Y);$   
 $T_3: r_3(X); r_3(Y); w_3(Y);$   
 $S_1: r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y);$   
 $S_2: r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); w_2(Z); w_3(Y); w_2(Y);$

**20.24.** Consider schedules  $S_3$ ,  $S_4$ , and  $S_5$  below. Determine whether each schedule is strict, cascadeless, recoverable, or nonrecoverable. (Determine the strictest recoverability condition that each schedule satisfies.)

- $S_3: r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); c_1; w_3(Y); c_3; r_2(Y);$   
 $w_2(Z); w_2(Y); c_2;$   
 $S_4: r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z);$   
 $w_2(Y); c_1; c_2; c_3;$   
 $S_5: r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); c_1; w_2(Z); w_3(Y);$   
 $w_2(Y); c_3; c_2;$

## Selected Bibliography

The concept of serializability and related ideas to maintain consistency in a database were introduced in Gray et al. (1975). The concept of the database transaction was first discussed in Gray (1981). Gray won the coveted ACM Turing Award in 1998 for his work on database transactions and implementation of transactions in relational DBMSs. Bernstein, Hadzilacos, and Goodman (1988) focus on concurrency control and recovery techniques in both centralized and distributed database systems; it is an excellent reference. Papadimitriou (1986) offers a more theoretical perspective. A large reference book of more than a thousand pages by Gray and Reuter (1993) offers a more practical perspective of transaction processing concepts and techniques. Elmagarmid (1992) offers collections of research papers on transaction processing for advanced applications. Transaction support in SQL is described in Date and Darwen (1997). View serializability is defined in Yannakakis (1984). Recoverability of schedules and reliability in databases is discussed in Hadzilacos (1983, 1988). Buffer replacement policies are discussed in Chou and DeWitt (1985). Snapshot isolation is discussed in Ports and Grittner (2012).

This page intentionally left blank

## Concurrency Control Techniques

In this chapter, we discuss a number of concurrency control techniques that are used to ensure the noninterference or isolation property of concurrently executing transactions. Most of these techniques ensure serializability of schedules—which we defined in Section 21.5—using **concurrency control protocols** (sets of rules) that guarantee serializability. One important set of protocols—known as *two-phase locking protocols*—employs the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently; a number of locking protocols are described in Sections 21.1 and 21.3.2. Locking protocols are used in some commercial DBMSs, but they are considered to have high overhead. Another set of concurrency control protocols uses **timestamps**. A timestamp is a unique identifier for each transaction, generated by the system. Timestamp values are generated in the same order as the transaction start times. Concurrency control protocols that use timestamp ordering to ensure serializability are introduced in Section 21.2. In Section 21.3, we discuss **multiversion** concurrency control protocols that use multiple versions of a data item. One multiversion protocol extends timestamp order to multiversion timestamp ordering (Section 21.3.1), and another extends timestamp order to two-phase locking (Section 21.3.2). In Section 21.4, we present a protocol based on the concept of **validation** or **certification** of a transaction after it executes its operations; these are sometimes called **optimistic protocols**, and they also assume that multiple versions of a data item can exist. In Section 21.4, we discuss a protocol that is based on the concept of **snapshot isolation**, which can utilize techniques similar to those proposed in validation-based and multiversion methods; these protocols are used in a number of commercial DBMSs and in certain cases are considered to have lower overhead than locking-based protocols.



Another factor that affects concurrency control is the **granularity** of the data items—that is, what portion of the database a data item represents. An item can be as small as a single attribute (field) value or as large as a disk block, or even a whole file or the entire database. We discuss granularity of items and a multiple granularity concurrency control protocol, which is an extension of two-phase locking, in Section 21.5. In Section 21.6, we describe concurrency control issues that arise when indexes are used to process transactions, and in Section 21.7 we discuss some additional concurrency control concepts. Section 21.8 summarizes the chapter.

It is sufficient to read Sections 21.1, 21.5, 21.6, and 21.7, and possibly 21.3.2, if your main interest is an introduction to the concurrency control techniques that are based on locking.

## 21.1 Two-Phase Locking Techniques for Concurrency Control

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items. In Section 21.1.1, we discuss the nature and types of locks. Then, in Section 21.1.2, we present protocols that use locking to guarantee serializability of transaction schedules. Finally, in Section 21.1.3, we describe two problems associated with the use of locks—deadlock and starvation—and show how these problems are handled in concurrency control protocols.

### 21.1.1 Types of Locks and System Lock Tables

Several types of locks are used in concurrency control. To introduce locking concepts gradually, first we discuss binary locks, which are simple but are also *too restrictive for database concurrency control purposes* and so are not used much. Then we discuss *shared/exclusive* locks—also known as *read/write* locks—which provide more general locking capabilities and are used in database locking schemes. In Section 21.3.2, we describe an additional type of lock called a *certify lock*, and we show how it can be used to improve performance of locking protocols.

**Binary Locks.** A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item  $X$ . If the value of the lock on  $X$  is 1, item  $X$  *cannot be accessed* by a database operation that requests the item. If the value of the lock on  $X$  is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item  $X$  as **lock( $X$ )**.

Two operations, `lock_item` and `unlock_item`, are used with binary locking. A transaction requests access to an item  $X$  by first issuing a **lock\_item( $X$ )** operation. If

```

lock_item(X):
B:  if LOCK(X) = 0          (*item is unlocked*)
      then LOCK(X) ← 1      (*lock the item*)
      else
        begin
        wait (until LOCK(X) = 0
              and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;            (* unlock the item *)
    if any transactions are waiting
    then wakeup one of the waiting transactions;

```

**Figure 21.1**

Lock and unlock operations for binary locks.

LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an **unlock\_item(X)** operation, which sets LOCK(X) back to 0 (**unlocks** the item) so that X may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item. A description of the lock\_item(X) and unlock\_item(X) operations is shown in Figure 21.1.

Notice that the lock\_item and unlock\_item operations must be implemented as indivisible units (known as **critical sections** in operating systems); that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits. In Figure 21.1, the wait command within the lock\_item(X) operation is usually implemented by putting the transaction in a waiting queue for item X until X is unlocked and the transaction can be granted access to it. Other transactions that also want to access X are placed in the same queue. Hence, the wait command is considered to be outside the lock\_item operation.

It is simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database. In its simplest form, each lock can be a record with three fields: <Data\_item\_name, LOCK, Locking\_transaction> plus a queue for transactions that are waiting to access the item. The system needs to maintain *only these records for the items that are currently locked* in a **lock table**, which could be organized as a hash file on the item name. Items not in the lock table are considered to be unlocked. The DBMS has a **lock manager subsystem** to keep track of and control access to locks.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction *T* must issue the operation lock\_item(X) before any read\_item(X) or write\_item(X) operations are performed in *T*.
2. A transaction *T* must issue the operation unlock\_item(X) after all read\_item(X) and write\_item(X) operations are completed in *T*.

3. A transaction  $T$  will not issue a `lock_item(X)` operation if it already holds the lock on item  $X$ .<sup>1</sup>
4. A transaction  $T$  will not issue an `unlock_item(X)` operation unless it already holds the lock on item  $X$ .

These rules can be enforced by the lock manager module of the DBMS. Between the `lock_item(X)` and `unlock_item(X)` operations in transaction  $T$ ,  $T$  is said to **hold the lock** on item  $X$ . At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

**Shared/Exclusive (or Read/Write) Locks.** The preceding binary locking scheme is too restrictive for database items because at most one transaction can hold a lock on a given item. We should allow several transactions to access the same item  $X$  if they all access  $X$  for *reading purposes only*. This is because read operations on the same item by different transactions are *not conflicting* (see Section 21.4.1). However, if a transaction is to write an item  $X$ , it must have exclusive access to  $X$ . For this purpose, a different type of lock, called a **multiple-mode lock**, is used. In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: `read_lock(X)`, `write_lock(X)`, and `unlock(X)`. A lock associated with an item  $X$ , `LOCK(X)`, now has three possible states: *read-locked*, *write-locked*, or *unlocked*. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

One method for implementing the preceding operations on a read/write lock is to keep track of the number of transactions that hold a shared (read) lock on an item in the lock table, as well as a list of transaction ids that hold a shared lock. Each record in the lock table will have four fields: `<Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>`. The system needs to maintain lock records only for locked items in the lock table. The value (state) of `LOCK` is either *read-locked* or *write-locked*, suitably coded (if we assume no records are kept in the lock table for unlocked items). If `LOCK(X) = write-locked`, the value of `locking_transaction(s)` is a *single transaction* that holds the exclusive (write) lock on  $X$ . If `LOCK(X) = read-locked`, the value of `locking_transaction(s)` is a list of one or more transactions that hold the shared (read) lock on  $X$ . The three operations `read_lock(X)`, `write_lock(X)`, and `unlock(X)` are described in Figure 21.2.<sup>2</sup> As before, each of the three locking operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed in a waiting queue for the item.

<sup>1</sup>This rule may be removed if we modify the `lock_item(X)` operation in Figure 21.1 so that if the item is currently locked by the requesting transaction, the lock is granted.

<sup>2</sup>These algorithms do not allow *upgrading* or *downgrading* of locks, as described later in this section. The reader can extend the algorithms to allow these additional operations.

**read\_lock(X):**

```

B:  if LOCK(X) = "unlocked"
      then begin LOCK(X) ← "read-locked";
           no_of_reads(X) ← 1
           end
      else if LOCK(X) = "read-locked"
           then no_of_reads(X) ← no_of_reads(X) + 1
      else begin
           wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
           go to B
           end;

```

**write\_lock(X):**

```

B:  if LOCK(X) = "unlocked"
      then LOCK(X) ← "write-locked"
      else begin
           wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
           go to B
           end;

```

**unlock (X):**

```

if LOCK(X) = "write-locked"
  then begin LOCK(X) ← "unlocked";
       wakeup one of the waiting transactions, if any
       end
else if LOCK(X) = "read-locked"
  then begin
       no_of_reads(X) ← no_of_reads(X) - 1;
       if no_of_reads(X) = 0
         then begin LOCK(X) = "unlocked";
              wakeup one of the waiting transactions, if any
              end
       end;

```

**Figure 21.2**

Locking and unlocking operations for two-mode (read/write, or shared/exclusive) locks.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction  $T$  must issue the operation  $\text{read\_lock}(X)$  or  $\text{write\_lock}(X)$  before any  $\text{read\_item}(X)$  operation is performed in  $T$ .
2. A transaction  $T$  must issue the operation  $\text{write\_lock}(X)$  before any  $\text{write\_item}(X)$  operation is performed in  $T$ .
3. A transaction  $T$  must issue the operation  $\text{unlock}(X)$  after all  $\text{read\_item}(X)$  and  $\text{write\_item}(X)$  operations are completed in  $T$ .<sup>3</sup>

<sup>3</sup>This rule may be relaxed to allow a transaction to unlock an item, then lock it again later. However, two-phase locking does not allow this.

4. A transaction  $T$  will not issue a `read_lock(X)` operation if it already holds a read (shared) lock or a write (exclusive) lock on item  $X$ . This rule may be relaxed for downgrading of locks, as we discuss shortly.
5. A transaction  $T$  will not issue a `write_lock(X)` operation if it already holds a read (shared) lock or write (exclusive) lock on item  $X$ . This rule may also be relaxed for upgrading of locks, as we discuss shortly.
6. A transaction  $T$  will not issue an `unlock(X)` operation unless it already holds a read (shared) lock or a write (exclusive) lock on item  $X$ .

**Conversion (Upgrading, Downgrading) of Locks.** It is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item  $X$  is allowed under certain conditions to **convert** the lock from one locked state to another. For example, it is possible for a transaction  $T$  to issue a `read_lock(X)` and then later to **upgrade** the lock by issuing a `write_lock(X)` operation. If  $T$  is the only transaction holding a read lock on  $X$  at the time it issues the `write_lock(X)` operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction  $T$  to issue a `write_lock(X)` and then later to **downgrade** the lock by issuing a `read_lock(X)` operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the `locking_transaction(s)` field) to store the information on which transactions hold locks on the item. The descriptions of the `read_lock(X)` and `write_lock(X)` operations in Figure 21.2 must be changed appropriately to allow for lock upgrading and downgrading. We leave this as an exercise for the reader.

Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 21.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 21.3(a) the items  $Y$  in  $T_1$  and  $X$  in  $T_2$  were unlocked too early. This allows a schedule such as the one shown in Figure 21.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction. The best-known protocol, two-phase locking, is described in the next section.

### 21.1.2 Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (`read_lock`, `write_lock`) precede the *first* unlock operation in the transaction.<sup>4</sup> Such a transaction can be divided into two phases: an **expanding or growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the

<sup>4</sup>This is unrelated to the two-phase commit protocol for recovery in distributed databases (see Chapter 23).

$T_1$	$T_2$
read_lock( $Y$ ); read_item( $Y$ ); unlock( $Y$ ); write_lock( $X$ ); read_item( $X$ ); $X := X + Y$ ; write_item( $X$ ); unlock( $X$ );	read_lock( $X$ ); read_item( $X$ ); unlock( $X$ ); write_lock( $Y$ ); read_item( $Y$ ); $Y := X + Y$ ; write_item( $Y$ ); unlock( $Y$ );

Result of serial schedule  $T_2$   
followed by  $T_1$ :  $X=70, Y=50$

	$T_1$	$T_2$
Time ↓	read_lock(Y); read_item(Y); unlock(Y);	read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); $Y := X + Y$ ; write_item(Y); unlock(Y);
	write_lock(X); read_item(X); $X := X + Y$ ; write_item(X); unlock(X);	

Transactions that do not obey two-phase locking. (a) Two transactions  $T_1$  and  $T_2$ . (b) Results of possible serial schedules of  $T_1$  and  $T_2$ . (c) A nonserializable schedule  $S$  that uses locks.

Transactions  $T_1$  and  $T_2$  in Figure 21.3(a) do not follow the two-phase locking protocol because the `write_lock(X)` operation follows the `unlock(Y)` operation in  $T_1$ , and similarly the `write_lock(Y)` operation follows the `unlock(X)` operation in  $T_2$ . If we enforce two-phase locking, the transactions can be rewritten as  $T_1'$  and  $T_2'$ , as shown in Figure 21.4. Now, the schedule shown in Figure 21.3(c) is not permitted for  $T_1'$  and  $T_2'$  (with their modified order of locking and unlocking operations) under the rules of locking described in Section 21.1.1 because  $T_1'$  will issue its `write_lock(X)` *before* it unlocks item  $Y$ ; consequently, when  $T_2'$  issues its `read_lock(X)`, it is forced to wait until  $T_1'$  releases the lock by issuing an `unlock(X)` in the schedule. However, this can lead to deadlock (see Section 21.1.3).

**Figure 21.4**  
 Transactions  $T_1'$  and  $T_2'$ , which are the same as  $T_1$  and  $T_2$  in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

$T_1'$	$T_2'$
read_lock(Y); read_item(Y); write_lock(X); unlock(Y) read_item(X); $X := X + Y$ ; write_item(X); unlock(X);	read_lock(X); read_item(X); write_lock(Y); unlock(X) read_item(Y); $Y := X + Y$ ; write_item(Y); unlock(Y);

It can be proved that, if *every* transaction in a schedule follows the two-phase locking protocol, the schedule is *guaranteed to be serializable*, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.

Two-phase locking may limit the amount of concurrency that can occur in a schedule because a transaction  $T$  may not be able to release an item  $X$  after it is through using it if  $T$  must lock an additional item  $Y$  later; or, conversely,  $T$  must lock the additional item  $Y$  before it needs it so that it can release  $X$ . Hence,  $X$  must remain locked by  $T$  until all items that the transaction needs to read or write have been locked; only then can  $X$  be released by  $T$ . Meanwhile, another transaction seeking to access  $X$  may be forced to wait, even though  $T$  is done with  $X$ ; conversely, if  $Y$  is locked earlier than it is needed, another transaction seeking to access  $Y$  is forced to wait even though  $T$  is not using  $Y$  yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit *all possible* serializable schedules (that is, some serializable schedules will be prohibited by the protocol).

**Basic, Conservative, Strict, and Rigorous Two-Phase Locking.** There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. Recall from Section 21.1.2 that the **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a *deadlock-free protocol*, as we will see in Section 21.1.3 when we discuss the deadlock problem. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in some situations.

In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules (see Section 21.4). In this variation, a transaction  $T$  does not release any



of its exclusive (write) locks until *after* it commits or aborts. Hence, no other transaction can read or write an item that is written by  $T$  unless  $T$  has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free. A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction  $T$  does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

Notice the difference between strict and rigorous 2PL: the former holds write-locks until it commits, whereas the latter holds all locks (read and write). Also, the difference between conservative and rigorous 2PL is that the former must lock all its items *before it starts*, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until *after it terminates* (by committing or aborting), so the transaction is in its expanding phase until it ends.

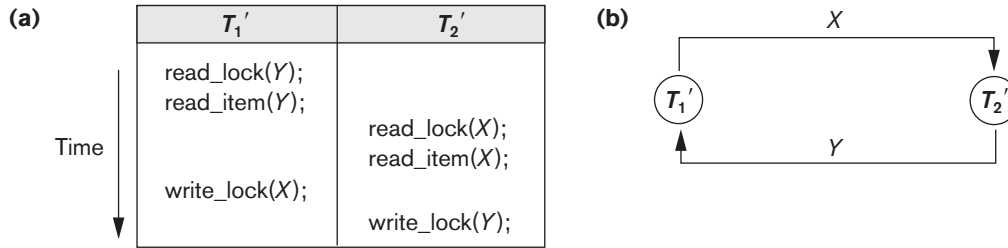
Usually the **concurrency control subsystem** itself is responsible for generating the `read_lock` and `write_lock` requests. For example, suppose the system is to enforce the strict 2PL protocol. Then, whenever transaction  $T$  issues a `read_item(X)`, the system calls the `read_lock(X)` operation on behalf of  $T$ . If the state of `LOCK(X)` is `write_locked` by some other transaction  $T'$ , the system places  $T$  in the waiting queue for item  $X$ ; otherwise, it grants the `read_lock(X)` request and permits the `read_item(X)` operation of  $T$  to execute. On the other hand, if transaction  $T$  issues a `write_item(X)`, the system calls the `write_lock(X)` operation on behalf of  $T$ . If the state of `LOCK(X)` is `write_locked` or `read_locked` by some other transaction  $T'$ , the system places  $T$  in the waiting queue for item  $X$ ; if the state of `LOCK(X)` is `read_locked` and  $T$  itself is the only transaction holding the read lock on  $X$ , the system upgrades the lock to `write_locked` and permits the `write_item(X)` operation by  $T$ . Finally, if the state of `LOCK(X)` is `unlocked`, the system grants the `write_lock(X)` request and permits the `write_item(X)` operation to execute. After each action, the system must *update its lock table* appropriately.

Locking is generally considered to have a high overhead, because every read or write operation is preceded by a system locking request. The use of locks can also cause two additional problems: deadlock and starvation. We discuss these problems and their solutions in the next section.

### 21.1.3 Dealing with Deadlock and Starvation

**Deadlock** occurs when *each* transaction  $T$  in a set of *two or more transactions* is waiting for some item that is locked by some other transaction  $T'$  in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock. A simple example is shown in Figure 21.5(a), where the two transactions  $T_1'$  and  $T_2'$  are deadlocked in a partial schedule;  $T_1'$  is in the waiting queue for  $X$ , which is locked by  $T_2'$ , whereas  $T_2'$  is in the waiting queue for  $Y$ , which is locked by  $T_1'$ . Meanwhile, neither  $T_1'$  nor  $T_2'$  nor any other transaction can access items  $X$  and  $Y$ .





**Figure 21.5**

Illustrating the deadlock problem. (a) A partial schedule of  $T_1'$  and  $T_2'$  that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

**Deadlock Prevention Protocols.** One way to prevent deadlock is to use a **deadlock prevention protocol**.<sup>5</sup> One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance* (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously, this solution further limits concurrency. A second protocol, which also limits concurrency, involves *ordering all the items* in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction? Some of these techniques use the concept of **transaction timestamp**  $TS(T)$ , which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which transactions are started; hence, if transaction  $T_1$  starts before transaction  $T_2$ , then  $TS(T_1) < TS(T_2)$ . Notice that the *older* transaction (which starts first) has the *smaller* timestamp value. Two schemes that prevent deadlock are called *wait-die* and *wound-wait*. Suppose that transaction  $T_i$  tries to lock an item  $X$  but is not able to because  $X$  is locked by some other transaction  $T_j$  with a conflicting lock. The rules followed by these schemes are:

- **Wait-die.** If  $TS(T_i) < TS(T_j)$ , then ( $T_i$  older than  $T_j$ )  $T_i$  is allowed to wait; otherwise ( $T_i$  younger than  $T_j$ ) abort  $T_i$  ( $T_i$  dies) and restart it later *with the same timestamp*.
- **Wound-wait.** If  $TS(T_i) < TS(T_j)$ , then ( $T_i$  older than  $T_j$ ) abort  $T_j$  ( $T_i$  wounds  $T_j$ ) and restart it later *with the same timestamp*; otherwise ( $T_i$  younger than  $T_j$ )  $T_i$  is allowed to wait.

<sup>5</sup>These protocols are not generally used in practice, either because of unrealistic assumptions or because of their possible overhead. Deadlock detection and timeouts (covered in the following sections) are more practical.

In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it. Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing. It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait for younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created. However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may *never actually cause a deadlock*.

Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the **no waiting algorithm**, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly. The **cautious waiting** algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction  $T_i$  tries to lock an item  $X$  but is not able to do so because  $X$  is locked by some other transaction  $T_j$  with a conflicting lock. The cautious waiting rule is as follows:

- **Cautious waiting.** If  $T_j$  is not blocked (not waiting for some other locked item), then  $T_i$  is blocked and allowed to wait; otherwise abort  $T_i$ .

It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction. By considering the time  $b(T)$  at which each blocked transaction  $T$  was blocked, if the two transactions  $T_i$  and  $T_j$  above both become blocked and  $T_i$  is waiting for  $T_j$ , then  $b(T_i) < b(T_j)$ , since  $T_i$  can only wait for  $T_j$  at a time when  $T_j$  is not blocked itself. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

**Deadlock Detection.** An alternative approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists. This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light. On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is heavy, it may be advantageous to use a deadlock prevention scheme.

A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction  $T_i$  is waiting to lock an item  $X$  that is currently locked by a transaction  $T_j$ , a directed edge ( $T_i \rightarrow T_j$ ) is created in the wait-for graph. When  $T_j$  releases the lock(s) on the items that  $T_i$  was

waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. One problem with this approach is the matter of determining *when* the system should check for a deadlock. One possibility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead. Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle. Figure 21.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 21.5(a).

If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as **victim selection**. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

**Timeouts.** Another simple scheme to deal with deadlock is the use of **timeouts**. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists.

**Starvation.** Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution. The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem. The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

## 21.2 Concurrency Control Based on Timestamp Ordering

The use of locking, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted

because of the deadlock problem. A different approach to concurrency control involves using transaction timestamps to order transaction execution for an equivalent serial schedule. In Section 21.2.1, we discuss timestamps; and in Section 21.2.2, we discuss how serializability is enforced by ordering conflicting operations in different transactions based on the transaction timestamps.

### 21.2.1 Timestamps

Recall that a **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*. We will refer to the timestamp of transaction  $T$  as  $TS(T)$ . Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

### 21.2.2 The Timestamp Ordering Algorithm for Concurrency Control

The idea for this scheme is to enforce the equivalent serial order on the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only equivalent serial schedule permitted* has the transactions in order of their timestamp values. This is called **timestamp ordering (TO)**. Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps. The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of *conflicting operations* in the schedule, the order in which the item is accessed must follow the timestamp order. To do this, the algorithm associates with each database item  $X$  two timestamp (TS) values:

1. **read\_TS(X)**. The **read timestamp** of item  $X$  is the largest timestamp among all the timestamps of transactions that have successfully read item  $X$ —that is,  $read\_TS(X) = TS(T)$ , where  $T$  is the *youngest* transaction that has read  $X$  successfully.
2. **write\_TS(X)**. The **write timestamp** of item  $X$  is the largest of all the timestamps of transactions that have successfully written item  $X$ —that is,  $write\_TS(X) = TS(T)$ , where  $T$  is the *youngest* transaction that has written  $X$  successfully. Based on the algorithm,  $T$  will also be the last transaction to write item  $X$ , as we shall see.

**Basic Timestamp Ordering (TO).** Whenever some transaction  $T$  tries to issue a  $\text{read\_item}(X)$  or a  $\text{write\_item}(X)$  operation, the **basic TO** algorithm compares the timestamp of  $T$  with  $\text{read\_TS}(X)$  and  $\text{write\_TS}(X)$  to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction  $T$  is aborted and resubmitted to the system as a new transaction with a *new timestamp*. If  $T$  is aborted and rolled back, any transaction  $T_1$  that may have used a value written by  $T$  must also be rolled back. Similarly, any transaction  $T_2$  that may have used a value written by  $T_1$  must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable. An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict. We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction  $T$  issues a  $\text{write\_item}(X)$  operation, the following check is performed:
  - a. If  $\text{read\_TS}(X) > \text{TS}(T)$  or if  $\text{write\_TS}(X) > \text{TS}(T)$ , then abort and roll back  $T$  and reject the operation. This should be done because some *younger* transaction with a timestamp greater than  $\text{TS}(T)$ —and hence *after*  $T$  in the timestamp ordering—has already read or written the value of item  $X$  before  $T$  had a chance to write  $X$ , thus violating the timestamp ordering.
  - b. If the condition in part (a) does not occur, then execute the  $\text{write\_item}(X)$  operation of  $T$  and set  $\text{write\_TS}(X)$  to  $\text{TS}(T)$ .
2. Whenever a transaction  $T$  issues a  $\text{read\_item}(X)$  operation, the following check is performed:
  - a. If  $\text{write\_TS}(X) > \text{TS}(T)$ , then abort and roll back  $T$  and reject the operation. This should be done because some younger transaction with timestamp greater than  $\text{TS}(T)$ —and hence *after*  $T$  in the timestamp ordering—has already written the value of item  $X$  before  $T$  had a chance to read  $X$ .
  - b. If  $\text{write\_TS}(X) \leq \text{TS}(T)$ , then execute the  $\text{read\_item}(X)$  operation of  $T$  and set  $\text{read\_TS}(X)$  to the *larger* of  $\text{TS}(T)$  and the current  $\text{read\_TS}(X)$ .

Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

**Strict Timestamp Ordering (TO).** A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable. In this variation, a transaction  $T$  issues a  $\text{read\_item}(X)$  or  $\text{write\_item}(X)$  such that  $\text{TS}(T) > \text{write\_TS}(X)$  has its read or write operation *delayed* until the transaction  $T'$  that *wrote* the value of  $X$  (hence  $\text{TS}(T') = \text{write\_TS}(X)$ ) has committed or aborted.

To implement this algorithm, it is necessary to simulate the locking of an item  $X$  that has been written by transaction  $T'$  until  $T'$  is either committed or aborted. This algorithm *does not cause deadlock*, since  $T$  waits for  $T'$  only if  $TS(T) > TS(T')$ .

**Thomas's Write Rule.** A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the `write_item(X)` operation as follows:

1. If  $read\_TS(X) > TS(T)$ , then abort and roll back  $T$  and reject the operation.
2. If  $write\_TS(X) > TS(T)$ , then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than  $TS(T)$ —and hence after  $T$  in the timestamp ordering—has already written the value of  $X$ . Thus, we must ignore the `write_item(X)` operation of  $T$  because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the `write_item(X)` operation of  $T$  and set  $write\_TS(X)$  to  $TS(T)$ .

## 21.3 Multiversion Concurrency Control Techniques

These protocols for concurrency control keep copies of the old values of a data item when the item is updated (written); they are known as **multiversion concurrency control** because several versions (values) of an item are kept by the system. When a transaction requests to read an item, the *appropriate* version is chosen to maintain the serializability of the currently executing schedule. One reason for keeping multiple versions is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version(s) of the item is retained. Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. In some cases, older versions can be kept in a temporary store. It is also possible that older versions may have to be maintained anyway—for example, for recovery purposes. Some database applications may require older versions to be kept to maintain a history of the changes of data item values. The extreme case is a *temporal database* (see Section 26.2), which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

Several multiversion concurrency control schemes have been proposed. We discuss two schemes here, one based on timestamp ordering and the other based on 2PL. In addition, the validation concurrency control method (see Section 21.4) also maintains multiple versions, and the *snapshot isolation* technique used in



several commercial systems (see Section 21.4) can be implemented by keeping older versions of items in a temporary store.

### 21.3.1 Multiversion Technique Based on Timestamp Ordering

In this method, several versions  $X_1, X_2, \dots, X_k$  of each data item  $X$  are maintained. For *each version*, the value of version  $X_i$  and the following two timestamps associated with version  $X_i$  are kept:

1. **read\_TS( $X_i$ )**. The **read timestamp** of  $X_i$  is the largest of all the timestamps of transactions that have successfully read version  $X_i$ .
2. **write\_TS( $X_i$ )**. The **write timestamp** of  $X_i$  is the timestamp of the transaction that wrote the value of version  $X_i$ .

Whenever a transaction  $T$  is allowed to execute a `write_item(X)` operation, a new version  $X_{k+1}$  of item  $X$  is created, with both the `write_TS( $X_{k+1}$ )` and the `read_TS( $X_{k+1}$ )` set to `TS( $T$ )`. Correspondingly, when a transaction  $T$  is allowed to read the value of version  $X_i$ , the value of `read_TS( $X_i$ )` is set to the larger of the current `read_TS( $X_i$ )` and `TS( $T$ )`.

To ensure serializability, the following rules are used:

1. If transaction  $T$  issues a `write_item(X)` operation, and version  $i$  of  $X$  has the highest `write_TS( $X_i$ )` of all versions of  $X$  that is also *less than or equal to* `TS( $T$ )`, and `read_TS( $X_i$ )`  $>$  `TS( $T$ )`, then abort and roll back transaction  $T$ ; otherwise, create a new version  $X_j$  of  $X$  with `read_TS( $X_j$ )` = `write_TS( $X_j$ )` = `TS( $T$ )`.
2. If transaction  $T$  issues a `read_item(X)` operation, find the version  $i$  of  $X$  that has the highest `write_TS( $X_i$ )` of all versions of  $X$  that is also *less than or equal to* `TS( $T$ )`; then return the value of  $X_i$  to transaction  $T$ , and set the value of `read_TS( $X_i$ )` to the larger of `TS( $T$ )` and the current `read_TS( $X_i$ )`.

As we can see in case 2, a `read_item(X)` is *always successful*, since it finds the appropriate version  $X_i$  to read based on the `write_TS` of the various existing versions of  $X$ . In case 1, however, transaction  $T$  may be aborted and rolled back. This happens if  $T$  attempts to write a version of  $X$  that should have been read by another transaction  $T'$  whose timestamp is `read_TS( $X_i$ )`; however,  $T'$  has already read version  $X_i$ , which was written by the transaction with timestamp equal to `write_TS( $X_i$ )`. If this conflict occurs,  $T$  is rolled back; otherwise, a new version of  $X$ , written by transaction  $T$ , is created. Notice that if  $T$  is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction  $T$  should not be allowed to commit until after all the transactions that have written some version that  $T$  has read have committed.

### 21.3.2 Multiversion Two-Phase Locking Using Certify Locks

In this multiple-mode locking scheme, there are *three locking modes* for an item—read, write, and *certify*—instead of just the two modes (read, write) discussed previously. Hence, the state of `LOCK(X)` for an item  $X$  can be one of read-locked, write-locked, certify-locked, or unlocked. In the standard locking scheme, with only read and write locks (see Section 21.1.1), a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme

(a)		Read	Write
Read	Yes	No	
Write	No	No	

(b)		Read	Write	Certify
Read	Yes	Yes	No	
Write	Yes	No	No	
Certify	No	No	No	

**Figure 21.6**

Lock compatibility tables.  
 (a) Lock compatibility table for read/write locking scheme.  
 (b) Lock compatibility table for read/write/certify locking scheme.

by means of the **lock compatibility table** shown in Figure 21.6(a). An entry of *Yes* means that if a transaction  $T$  holds the type of lock specified in the column header on item  $X$  and if transaction  $T'$  requests the type of lock specified in the row header on the same item  $X$ , then  $T'$  can obtain the lock because the locking modes are compatible. On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so  $T'$  must wait until  $T$  releases the lock.

In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions  $T'$  to read an item  $X$  while a single transaction  $T$  holds a write lock on  $X$ . This is accomplished by allowing *two versions* for each item  $X$ ; one version, the **committed version**, must always have been written by some committed transaction. The second **local version**  $X'$  can be created when a transaction  $T$  acquires a write lock on  $X$ . Other transactions can continue to read the *committed version* of  $X$  while  $T$  holds the write lock. Transaction  $T$  can write the value of  $X'$  as needed, without affecting the value of the committed version  $X$ . However, once  $T$  is ready to commit, it must obtain a **certify lock** on all items that it currently holds write locks on before it can commit; this is another form of **lock upgrading**. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks—which are exclusive locks—are acquired, the committed version  $X$  of the data item is set to the value of version  $X'$ , version  $X'$  is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure 21.6(b).

In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on *all the items* it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version  $X$  that was written by a committed transaction. However, deadlocks may occur, and these must be handled by variations of the techniques discussed in Section 21.1.3.



## 21.4 Validation (Optimistic) Techniques and Snapshot Isolation Concurrency Control

In all the concurrency control techniques we have discussed so far, a certain degree of checking is done *before* a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions.

In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing. Several concurrency control methods are based on the validation technique. We will describe only one scheme in Section 21.4.1. Then, in Section 21.4.2, we discuss concurrency control techniques that are based on the concept of **snapshot isolation**. The implementations of these concurrency control methods can utilize a combination of the concepts from validation-based techniques and versioning techniques, as well as utilizing timestamps. Some of these methods may suffer from anomalies that can violate serializability, but because they generally have lower overhead than 2PL, they have been implemented in several relational DBMSs.

### 21.4.1 Validation-Based (Optimistic) Concurrency Control

In this scheme, updates in the transaction are *not* applied directly to the database items on disk until the transaction reaches its end and is *validated*. During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction.<sup>6</sup> At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

1. **Read phase.** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase.** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase.** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation

---

<sup>6</sup>Note that this can be considered as keeping multiple versions of items!

phase is reached. If there is little interference among transactions, most will be validated successfully. However, if there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later; under such circumstances, optimistic techniques do not work well. The techniques are called *optimistic* because they assume that little interference will occur and hence most transaction will be validated successfully, so that there is no need to do checking during transaction execution. This assumption is generally true in many transaction processing workloads.

The optimistic protocol we describe uses transaction timestamps and also requires that the `write_sets` and `read_sets` of the transactions be kept by the system. Additionally, *start* and *end* times for the three phases need to be kept for each transaction. Recall that the `write_set` of a transaction is the set of items it writes, and the `read_set` is the set of items it reads. In the validation phase for transaction  $T_i$ , the protocol checks that  $T_i$  does not interfere with any recently committed transactions or with any other concurrent transactions that have started their validation phase. The validation phase for  $T_i$  checks that, for *each* such transaction  $T_j$  that is either recently committed or is in its validation phase, *one* of the following conditions holds:

1. Transaction  $T_j$  completes its write phase before  $T_i$  starts its read phase.
2.  $T_i$  starts its write phase after  $T_j$  completes its write phase, and the `read_set` of  $T_i$  has no items in common with the `write_set` of  $T_j$ .
3. Both the `read_set` and `write_set` of  $T_i$  have no items in common with the `write_set` of  $T_j$ , and  $T_j$  completes its read phase before  $T_i$  completes its read phase.

When validating transaction  $T_i$  against each one of the transactions  $T_j$ , the first condition is checked first since (1) is the simplest condition to check. Only if condition 1 is false is condition 2 checked, and only if (2) is false is condition 3—the most complex to evaluate—checked. If any one of these three conditions holds with each transaction  $T_j$ , there is no interference and  $T_i$  is validated successfully. If *none* of these three conditions holds for any one  $T_j$ , the validation of transaction  $T_i$  fails (because  $T_i$  and  $T_j$  may violate serializability) and so  $T_i$  is aborted and restarted later because interference with  $T_j$  may have occurred.

## 21.4.2 Concurrency Control Based on Snapshot Isolation

As we discussed in Section 20.6, the basic definition of **snapshot isolation** is that a transaction sees the data items that it reads based on the committed values of the items in the *database snapshot* (or database state) when the transaction starts. Snapshot isolation will ensure that the phantom record problem does not occur, since the database transaction, or, in some cases, the database statement, will only see the records that were committed in the database at the time the transaction started. Any insertions, deletions, or updates that occur after the transaction starts will not be seen by the transaction. In addition, snapshot isolation does not allow the problems of dirty read and nonrepeatable read to occur. However, certain anomalies that violate serializability can occur when snapshot isolation is used as the basis for

concurrency control. Although these anomalies are rare, they are very difficult to detect and may result in an inconsistent or corrupted database. The interested reader can refer to the end-of-chapter bibliography for papers that discuss in detail the rare types of anomalies that can occur.

In this scheme, read operations do not require read locks to be applied to the items, thus reducing the overhead associated with two-phase locking. However, write operations do require write locks. Thus, for transactions that have many reads, the performance is much better than 2PL. When writes do occur, the system will have to keep track of older versions of the updated items in a **temporary version store** (sometimes known as *tempstore*), with the timestamps of when the version was created. This is necessary so that a transaction that started before the item was written can still read the value (version) of the item that was in the database snapshot when the transaction started.

To keep track of versions, items that have been updated will have pointers to a list of recent versions of the item in the *tempstore*, so that the correct item can be read for each transaction. The tempstore items will be removed when no longer needed, so a method to decide when to remove unneeded versions will be needed.

Variations of this method have been used in several commercial and open source DBMSs, including Oracle and PostGRES. If the users require guaranteed serializability, then the problems with anomalies that violate serializability will have to be solved by the programmers/software engineers by analyzing the set of transactions to determine which types of anomalies can occur, and adding checks that do not permit these anomalies. This can place a burden on the software developers when compared to the DBMS enforcing serializability in all cases.

Variations of snapshot isolation (SI) techniques, known as **serializable snapshot isolation (SSI)**, have been proposed and implemented in some of the DBMSs that use SI as their primary concurrency control method. For example, recent versions of the PostGRES DBMS allow the user to choose between basic SI and SSI. The tradeoff is ensuring full serializability with SSI versus living with possible rare anomalies but having better performance with basic SI. The interested reader is referred to the end-of-chapter bibliography for more complete discussions of these topics.

## 21.5 Granularity of Data Items and Multiple Granularity Locking

All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:

- A database record
- A field value of a database record
- A disk block
- A whole file
- The whole database

The particular choice of data item type can affect the performance of concurrency control and recovery. In Section 21.5.1, we discuss some of the tradeoffs with regard to choosing the granularity level used for locking; and in Section 21.5.2, we discuss a multiple granularity locking scheme, where the granularity level (size of the data item) may be changed dynamically.

### 21.5.1 Granularity Level Considerations for Locking

The size of data items is often called the **data item granularity**. *Fine granularity* refers to small item sizes, whereas *coarse granularity* refers to large item sizes. Several tradeoffs must be considered in choosing the data item size. We will discuss data item size in the context of locking, although similar arguments can be made for other concurrency control techniques.

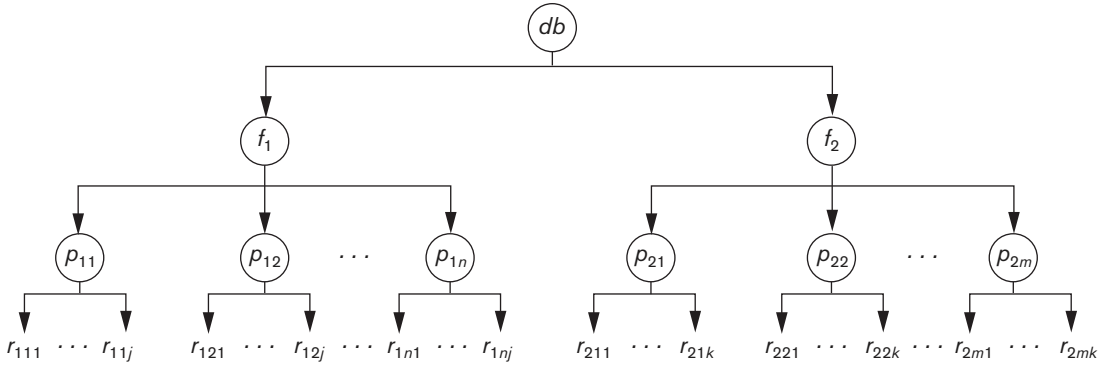
First, notice that the larger the data item size is, the lower the degree of concurrency permitted. For example, if the data item size is a disk block, a transaction  $T$  that needs to lock a single record  $B$  must lock the whole disk block  $X$  that contains  $B$  because a lock is associated with the whole data item (block). Now, if another transaction  $S$  wants to lock a different record  $C$  that happens to reside in the same disk block  $X$  in a conflicting lock mode, it is forced to wait. If the data item size was a single record instead of a disk block, transaction  $S$  would be able to proceed, because it would be locking a different data item (record).

On the other hand, the smaller the data item size is, the more the number of items in the database. Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager. More lock and unlock operations will be performed, causing a higher overhead. In addition, more storage space will be required for the lock table. For timestamps, storage is required for the read\_TS and write\_TS for each data item, and there will be similar overhead for handling a large number of items.

Given the above tradeoffs, an obvious question can be asked: What is the best item size? The answer is that it *depends on the types of transactions involved*. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record. On the other hand, if a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items.

### 21.5.2 Multiple Granularity Level Locking

Since the best granularity size depends on the given transaction, it seems appropriate that a database system should support multiple levels of granularity, where the granularity level can be adjusted dynamically for various mixes of transactions. Figure 21.7 shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records. This can be used to illustrate a **multiple granularity level** 2PL protocol, with shared/exclusive locking modes, where a lock can be requested at any level. However, additional types of locks will be needed to support such a protocol efficiently.

**Figure 21.7**

A granularity hierarchy for illustrating multiple granularity level locking.

Consider the following scenario, which refers to the example in Figure 21.7. Suppose transaction  $T_1$  wants to update *all the records* in file  $f_1$ , and  $T_1$  requests and is granted an exclusive lock for  $f_1$ . Then all of  $f_1$ 's pages ( $p_{11}$  through  $p_{1n}$ )—and the records contained on those pages—are locked in exclusive mode. This is beneficial for  $T_1$  because setting a single file-level lock is more efficient than setting  $n$  page-level locks or having to lock each record individually. Now suppose another transaction  $T_2$  only wants to read record  $r_{1nj}$  from page  $p_{1n}$  of file  $f_1$ ; then  $T_2$  would request a shared record-level lock on  $r_{1nj}$ . However, the database system (that is, the transaction manager or, more specifically, the lock manager) must verify the compatibility of the requested lock with already held locks. One way to verify this is to traverse the tree from the leaf  $r_{1nj}$  to  $p_{1n}$  to  $f_1$  to  $db$ . If at any time a conflicting lock is held on any of those items, then the lock request for  $r_{1nj}$  is denied and  $T_2$  is blocked and must wait. This traversal would be fairly efficient.

However, what if transaction  $T_2$ 's request came *before* transaction  $T_1$ 's request? In this case, the shared record lock is granted to  $T_2$  for  $r_{1nj}$ , but when  $T_1$ 's file-level lock is requested, it can be time-consuming for the lock manager to check all nodes (pages and records) that are descendants of node  $f_1$  for a lock conflict. This would be very inefficient and would defeat the purpose of having multiple granularity level locks.

To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed. The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants. There are three types of intention locks:

1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

**Figure 21.8**

Lock compatibility matrix for multiple granularity locking.

3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

The compatibility table of the three intention locks, and the actual shared and exclusive locks, is shown in Figure 21.8. In addition to the three types of intention locks, an appropriate locking protocol must be used. The **multiple granularity locking (MGL)** protocol consists of the following rules:

1. The lock compatibility (based on Figure 21.8) must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node  $N$  can be locked by a transaction  $T$  in S or IS mode only if the parent node  $N$  is already locked by transaction  $T$  in either IS or IX mode.
4. A node  $N$  can be locked by a transaction  $T$  in X, IX, or SIX mode only if the parent of node  $N$  is already locked by transaction  $T$  in either IX or SIX mode.
5. A transaction  $T$  can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
6. A transaction  $T$  can unlock a node,  $N$ , only if none of the children of node  $N$  are currently locked by  $T$ .

Rule 1 simply states that conflicting locks cannot be granted. Rules 2, 3, and 4 state the conditions when a transaction may lock a given node in any of the lock modes. Rules 5 and 6 of the MGL protocol enforce 2PL rules to produce serializable schedules. Basically, the locking *starts from the root* and goes down the tree until the node that needs to be locked is encountered, whereas unlocking *starts from the locked node* and goes up the tree until the root itself is unlocked. To illustrate the MGL protocol with the database hierarchy in Figure 21.7, consider the following three transactions:

1.  $T_1$  wants to update record  $r_{111}$  and record  $r_{211}$ .
2.  $T_2$  wants to update all records on page  $p_{12}$ .
3.  $T_3$  wants to read record  $r_{11j}$  and the entire  $f_2$  file.

<i>T</i> <sub>1</sub>	<i>T</i> <sub>2</sub>	<i>T</i> <sub>3</sub>
IX( <i>db</i> ) IX( <i>f</i> <sub>1</sub> )	IX( <i>db</i> )	IS( <i>db</i> ) IS( <i>f</i> <sub>1</sub> ) IS( <i>p</i> <sub>11</sub> )
IX( <i>p</i> <sub>11</sub> ) X( <i>r</i> <sub>111</sub> )	IX( <i>f</i> <sub>1</sub> ) X( <i>p</i> <sub>12</sub> )	S( <i>r</i> <sub>11j</sub> )
IX( <i>f</i> <sub>2</sub> ) IX( <i>p</i> <sub>21</sub> ) X( <i>p</i> <sub>211</sub> )		
unlock( <i>r</i> <sub>211</sub> ) unlock( <i>p</i> <sub>21</sub> ) unlock( <i>f</i> <sub>2</sub> )		S( <i>f</i> <sub>2</sub> )
unlock( <i>r</i> <sub>111</sub> ) unlock( <i>p</i> <sub>11</sub> ) unlock( <i>f</i> <sub>1</sub> ) unlock( <i>db</i> )	unlock( <i>p</i> <sub>12</sub> ) unlock( <i>f</i> <sub>1</sub> ) unlock( <i>db</i> )	
		unlock( <i>r</i> <sub>11j</sub> ) unlock( <i>p</i> <sub>11</sub> ) unlock( <i>f</i> <sub>1</sub> ) unlock( <i>f</i> <sub>2</sub> ) unlock( <i>db</i> )

**Figure 21.9**  
Lock operations to  
illustrate a serializable  
schedule.

Figure 21.9 shows a possible serializable schedule for these three transactions. Only the lock and unlock operations are shown. The notation <lock\_type>(<item>) is used to display the locking operations in the schedule.

The multiple granularity level protocol is especially suited when processing a mix of transactions that include (1) short transactions that access only a few items (records or fields) and (2) long transactions that access entire files. In this environment, less transaction blocking and less locking overhead are incurred by such a protocol when compared to a single-level granularity locking approach.

## 21.6 Using Locks for Concurrency Control in Indexes

Two-phase locking can also be applied to B-tree and B<sup>+</sup>-tree indexes (see Chapter 19), where the nodes of an index correspond to disk pages. However, holding locks on index pages until the shrinking phase of 2PL could cause an undue amount of transaction blocking because searching an index always *starts at the root*. For example, if a transaction wants to insert a record (write operation), the root would be locked in exclusive mode, so all other conflicting lock requests for the index must wait until the transaction enters its shrinking phase. This blocks all other transactions from accessing the index, so in practice other approaches to locking an index must be used.

The tree structure of the index can be taken advantage of when developing a concurrency control scheme. For example, when an index search (read operation) is being executed, a path in the tree is traversed from the root to a leaf. Once a lower-level node in the path has been accessed, the higher-level nodes in that path will not be used again. So once a read lock on a child node is obtained, the lock on the parent node can be released. When an insertion is being applied to a leaf node (that is, when a key and a pointer are inserted), then a specific leaf node must be locked in exclusive mode. However, if that node is not full, the insertion will not cause changes to higher-level index nodes, which implies that they need not be locked exclusively.

A conservative approach for insertions would be to lock the root node in exclusive mode and then to access the appropriate child node of the root. If the child node is not full, then the lock on the root node can be released. This approach can be applied all the way down the tree to the leaf, which is typically three or four levels from the root. Although exclusive locks are held, they are soon released. An alternative, more **optimistic approach** would be to request and hold *shared* locks on the nodes leading to the leaf node, with an *exclusive* lock on the leaf. If the insertion causes the leaf to split, insertion will propagate to one or more higher-level nodes. Then, the locks on the higher-level nodes can be upgraded to exclusive mode.

Another approach to index locking is to use a variant of the B<sup>+</sup>-tree, called the **B-link tree**. In a B-link tree, sibling nodes on the same level are linked at every level. This allows shared locks to be used when requesting a page and requires that the lock be released before accessing the child node. For an insert operation, the shared lock on a node would be upgraded to exclusive mode. If a split occurs, the parent node must be relocked in exclusive mode. One complication is for search operations executed concurrently with the update. Suppose that a concurrent update operation follows the same path as the search and inserts a new entry into the leaf node. Additionally, suppose that the insert causes that leaf node to split. When the insert is done, the search process resumes, following the pointer to the desired leaf, only to find that the key it is looking for is not present because the split has moved that key into a new leaf node, which would be the *right sibling* of the original leaf



node. However, the search process can still succeed if it follows the pointer (link) in the original leaf node to its right sibling, where the desired key has been moved.

Handling the deletion case, where two or more nodes from the index tree merge, is also part of the B-link tree concurrency protocol. In this case, locks on the nodes to be merged are held as well as a lock on the parent of the two nodes to be merged.

## 21.7 Other Concurrency Control Issues

In this section, we discuss some other issues relevant to concurrency control. In Section 21.7.1, we discuss problems associated with insertion and deletion of records and we revisit the *phantom problem*, which may occur when records are inserted. This problem was described as a potential problem requiring a concurrency control measure in Section 20.6. In Section 21.7.2, we discuss problems that may occur when a transaction outputs some data to a monitor before it commits, and then the transaction is later aborted.

### 21.7.1 Insertion, Deletion, and Phantom Records

When a new data item is **inserted** in the database, it obviously cannot be accessed until after the item is created and the insert operation is completed. In a locking environment, a lock for the item can be created and set to exclusive (write) mode; the lock can be released at the same time as other write locks would be released, based on the concurrency control protocol being used. For a timestamp-based protocol, the read and write timestamps of the new item are set to the timestamp of the creating transaction.

Next, consider a **deletion operation** that is applied on an existing data item. For locking protocols, again an exclusive (write) lock must be obtained before the transaction can delete the item. For timestamp ordering, the protocol must ensure that no later transaction has read or written the item before allowing the item to be deleted.

A situation known as the **phantom problem** can occur when a new record that is being inserted by some transaction  $T$  satisfies a condition that a set of records accessed by another transaction  $T'$  must satisfy. For example, suppose that transaction  $T$  is inserting a new EMPLOYEE record whose Dno = 5, whereas transaction  $T'$  is accessing all EMPLOYEE records whose Dno = 5 (say, to add up all their Salary values to calculate the personnel budget for department 5). If the equivalent serial order is  $T$  followed by  $T'$ , then  $T'$  must read the new EMPLOYEE record and include its Salary in the sum calculation. For the equivalent serial order  $T'$  followed by  $T$ , the new salary should not be included. Notice that although the transactions logically conflict, in the latter case there is really no record (data item) in common between the two transactions, since  $T'$  may have locked all the records with Dno = 5 *before*  $T$  inserted the new record. This is because the record that causes the conflict is a **phantom record** that has suddenly appeared in the database on being inserted. If other operations in the two transactions conflict, the conflict due to the phantom record may not be recognized by the concurrency control protocol.

One solution to the phantom record problem is to use **index locking**, as discussed in Section 21.6. Recall from Chapter 19 that an index includes entries that have an attribute value plus a set of pointers to all records in the file with that value. For example, an index on Dno of EMPLOYEE would include an entry for each distinct Dno value plus a set of pointers to all EMPLOYEE records with that value. If the index entry is locked before the record itself can be accessed, then the conflict on the phantom record can be detected, because transaction  $T'$  would request a read lock on the *index entry* for Dno = 5, and  $T$  would request a write lock on the same entry *before* it could place the locks on the actual records. Since the index locks conflict, the phantom conflict would be detected.

A more general technique, called **predicate locking**, would lock access to all records that satisfy an arbitrary *predicate* (condition) in a similar manner; however, predicate locks have proved to be difficult to implement efficiently. If the concurrency control method is based on snapshot isolation (see Section 21.4.2), then the transaction that reads the items will access the database snapshot at the time the transaction starts; any records inserted after that will not be retrieved by the transaction.

### 21.7.2 Interactive Transactions

Another problem occurs when interactive transactions read input and write output to an interactive device, such as a monitor screen, before they are committed. The problem is that a user can input a value of a data item to a transaction  $T$  that is based on some value written to the screen by transaction  $T'$ , which may not have committed. This dependency between  $T$  and  $T'$  cannot be modeled by the system concurrency control method, since it is only based on the user interacting with the two transactions.

An approach to dealing with this problem is to postpone output of transactions to the screen until they have committed.

### 21.7.3 Latches

Locks held for a short duration are typically called **latches**. Latches do not follow the usual concurrency control protocol such as two-phase locking. For example, a latch can be used to guarantee the physical integrity of a disk page when that page is being written from the buffer to disk. A latch would be acquired for the page, the page written to disk, and then the latch released.

## 21.8 Summary

In this chapter, we discussed DBMS techniques for concurrency control. We started in Section 21.1 by discussing lock-based protocols, which are commonly used in practice. In Section 21.1.2 we described the two-phase locking (2PL) protocol and a number of its variations: basic 2PL, strict 2PL, conservative 2PL, and rigorous 2PL. The strict and rigorous variations are more common because of

their better recoverability properties. We introduced the concepts of shared (read) and exclusive (write) locks (Section 21.1.1) and showed how locking can guarantee serializability when used in conjunction with the two-phase locking rule. We also presented various techniques for dealing with the deadlock problem in Section 21.1.3, which can occur with locking. In practice, it is common to use timeouts and deadlock detection (wait-for graphs). Deadlock prevention protocols, such as no waiting and cautious waiting, can also be used.

We then presented other concurrency control protocols. These include the timestamp ordering protocol (Section 21.2), which ensures serializability based on the order of transaction timestamps. Timestamps are unique, system-generated transaction identifiers. We discussed Thomas's write rule, which improves performance but does not guarantee serializability. The strict timestamp ordering protocol was also presented. We discussed two multiversion protocols (Section 21.3), which assume that older versions of data items can be kept in the database. One technique, called multiversion two-phase locking (which has been used in practice), assumes that two versions can exist for an item and attempts to increase concurrency by making write and read locks compatible (at the cost of introducing an additional certify lock mode). We also presented a multiversion protocol based on timestamp ordering. In Section 21.4.1, we presented an example of an optimistic protocol, which is also known as a certification or validation protocol.

We then discussed concurrency control methods that are based on the concept of snapshot isolation in Section 21.4.2; these are used in several DBMSs because of their lower overhead. The basic snapshot isolation method can allow nonserializable schedules in rare cases because of certain anomalies that are difficult to detect; these anomalies may cause a corrupted database. A variation known as serializable snapshot isolation has been recently developed and ensures serializable schedules.

Then in Section 21.5 we turned our attention to the important practical issue of data item granularity. We described a multigranularity locking protocol that allows the change of granularity (item size) based on the current transaction mix, with the goal of improving the performance of concurrency control. An important practical issue was then presented in Section 21.6, which is to develop locking protocols for indexes so that indexes do not become a hindrance to concurrent access. Finally, in Section 21.7, we introduced the phantom problem and problems with interactive transactions, and we briefly described the concept of latches and how this concept differs from locks.

## Review Questions

- 21.1.** What is the two-phase locking protocol? How does it guarantee serializability?
- 21.2.** What are some variations of the two-phase locking protocol? Why is strict or rigorous two-phase locking often preferred?
- 21.3.** Discuss the problems of deadlock and starvation, and the different approaches to dealing with these problems.

- 21.4. Compare binary locks to exclusive/shared locks. Why is the latter type of locks preferable?
- 21.5. Describe the wait-die and wound-wait protocols for deadlock prevention.
- 21.6. Describe the cautious waiting, no waiting, and timeout protocols for deadlock prevention.
- 21.7. What is a timestamp? How does the system generate timestamps?
- 21.8. Discuss the timestamp ordering protocol for concurrency control. How does strict timestamp ordering differ from basic timestamp ordering?
- 21.9. Discuss two multiversion techniques for concurrency control. What is a certify lock? What are the advantages and disadvantages of using certify locks?
- 21.10. How do optimistic concurrency control techniques differ from other concurrency control techniques? Why are they also called validation or certification techniques? Discuss the typical phases of an optimistic concurrency control method.
- 21.11. What is snapshot isolation? What are the advantages and disadvantages of concurrency control methods that are based on snapshot isolation?
- 21.12. How does the granularity of data items affect the performance of concurrency control? What factors affect selection of granularity size for data items?
- 21.13. What type of lock is needed for insert and delete operations?
- 21.14. What is multiple granularity locking? Under what circumstances is it used?
- 21.15. What are intention locks?
- 21.16. When are latches used?
- 21.17. What is a phantom record? Discuss the problem that a phantom record can cause for concurrency control.
- 21.18. How does index locking resolve the phantom problem?
- 21.19. What is a predicate lock?

## Exercises

- 21.20. Prove that the basic two-phase locking protocol guarantees conflict serializability of schedules. (*Hint*: Show that if a serializability graph for a schedule has a cycle, then at least one of the transactions participating in the schedule does not obey the two-phase locking protocol.)
- 21.21. Modify the data structures for multiple-mode locks and the algorithms for `read_lock(X)`, `write_lock(X)`, and `unlock(X)` so that upgrading and downgrading of locks are possible. (*Hint*: The lock needs to check the transaction id(s) that hold the lock, if any.)

- 21.22.** Prove that strict two-phase locking guarantees strict schedules.
- 21.23.** Prove that the wait-die and wound-wait protocols avoid deadlock and starvation.
- 21.24.** Prove that cautious waiting avoids deadlock.
- 21.25.** Apply the timestamp ordering algorithm to the schedules in Figures 21.8(b) and (c), and determine whether the algorithm will allow the execution of the schedules.
- 21.26.** Repeat Exercise 21.25, but use the multiversion timestamp ordering method.
- 21.27.** Why is two-phase locking not used as a concurrency control method for indexes such as  $B^+$ -trees?
- 21.28.** The compatibility matrix in Figure 21.8 shows that IS and IX locks are compatible. Explain why this is valid.
- 21.29.** The MGL protocol states that a transaction  $T$  can unlock a node  $N$ , only if none of the children of node  $N$  are still locked by transaction  $T$ . Show that without this condition, the MGL protocol would be incorrect.

## Selected Bibliography

The two-phase locking protocol and the concept of predicate locks were first proposed by Eswaran et al. (1976). Bernstein et al. (1987), Gray and Reuter (1993), and Papadimitriou (1986) focus on concurrency control and recovery. Kumar (1996) focuses on performance of concurrency control methods. Locking is discussed in Gray et al. (1975), Lien and Weinberger (1978), Kadem and Silbershatz (1980), and Korth (1983). Deadlocks and wait-for graphs were formalized by Holt (1972), and the wait-wound and wound-die schemes are presented in Rosenkrantz et al. (1978). Cautious waiting is discussed in Hsu and Zhang (1992). Helal et al. (1993) compares various locking approaches.

Timestamp-based concurrency control techniques are discussed in Bernstein and Goodman (1980) and Reed (1983). Optimistic concurrency control is discussed in Kung and Robinson (1981) and Bessiouni (1988). Papadimitriou and Kanellakis (1979) and Bernstein and Goodman (1983) discuss multiversion techniques. Multiversion timestamp ordering was proposed in Reed (1979, 1983), and multiversion two-phase locking is discussed in Lai and Wilkinson (1984). A method for multiple locking granularities was proposed in Gray et al. (1975), and the effects of locking granularities are analyzed in Ries and Stonebraker (1977). Bhargava and Reidl (1988) presents an approach for dynamically choosing among various concurrency control and recovery methods. Concurrency control methods for indexes are presented in Lehman and Yao (1981) and in Shasha and Goodman (1988). A performance study of various  $B^+$ -tree concurrency control algorithms is presented in Srinivasan and Carey (1991).

Anomalies that can occur with basic snapshot isolation are discussed in Fekete et al. (2004), Jorwekar et al. (2007), and Ports and Grittner (2012), among others. Modifying snapshot isolation to make it serializable is discussed in Cahill et al. (2008), Fekete et al. (2005), Revilak et al. (2011), and Ports and Grittner (2012).

Other work on concurrency control includes semantic-based concurrency control (Badrinath & Ramamritham, 1992), transaction models for long-running activities (Dayal et al., 1991), and multilevel transaction management (Hasse & Weikum, 1991).

This page intentionally left blank

## Database Recovery Techniques

In this chapter, we discuss some of the techniques that can be used for database recovery in case of system failure. In Section 20.1.4 we discussed the different causes of failure, such as system crashes and transaction errors. In Section 20.2, we introduced some of the concepts that are used by recovery processes, such as the system log and commit points.

This chapter presents additional concepts that are relevant to recovery protocols and provides an overview of the various database recovery algorithms. We start in Section 22.1 with an outline of a typical recovery procedure and a categorization of recovery algorithms, and then we discuss several recovery concepts, including write-ahead logging, in-place versus shadow updates, and the process of rolling back (undoing) the effect of an incomplete or failed transaction. In Section 22.2, we present recovery techniques based on *deferred update*, also known as the NO-UNDO/REDO technique, where the data on disk is not updated until *after* a transaction commits. In Section 22.3, we discuss recovery techniques based on *immediate update*, where data can be updated on disk during transaction execution; these include the UNDO/REDO and UNDO/NO-REDO algorithms. In Section 22.4, we discuss the technique known as shadowing or shadow paging, which can be categorized as a NO-UNDO/NO-REDO algorithm. An example of a practical DBMS recovery scheme, called ARIES, is presented in Section 22.5. Recovery in multidatabases is briefly discussed in Section 22.6. Finally, techniques for recovery from catastrophic failure are discussed in Section 22.7. Section 22.8 summarizes the chapter.

Our emphasis is on conceptually describing several different approaches to recovery. For descriptions of recovery features in specific systems, the reader should consult the bibliographic notes at the end of the chapter and the online and printed user manuals for those systems. Recovery techniques are often intertwined with the concurrency control mechanisms. Certain recovery techniques are best used with



specific concurrency control methods. We will discuss recovery concepts independently of concurrency control mechanisms.

## 22.1 Recovery Concepts

### 22.1.1 Recovery Outline and Categorization of Recovery Algorithms

Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the **system log**, as we discussed in Section 21.2.2. A typical strategy for recovery may be summarized informally as follows:

1. If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was *backed up* to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or *redoing* the operations of committed transactions from the *backed-up* log, up to the time of failure.
2. When the database on disk is not physically damaged, and a noncatastrophic failure of types 1 through 4 in Section 21.1.4 has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database. For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by *undoing* its write operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the database; for example, if a transaction has committed but some of its write operations have not yet been written to disk. For noncatastrophic failure, the recovery protocol does not need a complete archival copy of the database. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

Conceptually, we can distinguish two main policies for recovery from noncatastrophic transaction failures: deferred update and immediate update. The **deferred update** techniques do not physically update the database on disk until *after* a transaction commits; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains (the DBMS main memory cache; see Section 20.2.4). Before commit, the updates are recorded persistently in the log file on disk, and then after commit, the updates are written to the database from the main memory buffers. If a transaction fails before reaching its commit point, it will not have changed the database on disk in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed

transaction from the log, because their effect may not yet have been recorded in the database on disk. Hence, deferred update is also known as the **NO-UNDO/REDO algorithm**. We discuss this technique in Section 22.2.

In the **immediate update** techniques, the database *may be updated* by some operations of a transaction *before* the transaction reaches its commit point. However, these operations must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery still possible. If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both *undo* and *redo* may be required during recovery. This technique, known as the **UNDO/REDO algorithm**, requires both operations during recovery and is used most often in practice. A variation of the algorithm where all updates are required to be recorded in the database on disk *before* a transaction commits requires *undo* only, so it is known as the **UNDO/NO-REDO algorithm**. We discuss these two techniques in Section 22.3.

The UNDO and REDO operations are required to be **idempotent**—that is, executing an operation multiple times is equivalent to executing it just once. In fact, the whole recovery process should be idempotent because if the system were to fail during the recovery process, the next recovery attempt might UNDO and REDO certain *write\_item* operations that had already been executed during the first recovery process. The result of recovery from a system crash *during recovery* should be the same as the result of recovering *when there is no crash during recovery*!

### 22.1.2 Caching (Buffering) of Disk Blocks

The recovery process is often closely intertwined with operating system functions—in particular, the buffering of database disk pages in the DBMS main memory cache. Typically, multiple disk pages that include the data items to be updated are **cached** into main memory buffers and then updated in memory before being written back to disk. The caching of disk pages is traditionally an operating system function, but because of its importance to the efficiency of recovery procedures, it is handled by the DBMS by calling low-level operating systems routines (see Section 20.2.4).

In general, it is convenient to consider recovery in terms of the database disk pages (blocks). Typically a collection of in-memory buffers, called the **DBMS cache**, is kept under the control of the DBMS for the purpose of holding these buffers. A **directory** for the cache is used to keep track of which database items are in the buffers.<sup>1</sup> This can be a table of <Disk\_page\_address, Buffer\_location, ... > entries. When the DBMS requests action on some item, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache. If it is not,

---

<sup>1</sup>This is somewhat similar to the concept of page tables used by the operating system.

the item must be located on disk, and the appropriate disk pages are copied into the cache. It may be necessary to **replace** (or **flush**) some of the cache buffers to make space available for the new item (see Section 20.2.4).

The entries in the DBMS cache directory hold additional information relevant to buffer management. Associated with each buffer in the cache is a **dirty bit**, which can be included in the directory entry to indicate whether or not the buffer has been modified. When a page is first read from the database disk into a cache buffer, a new entry is inserted in the cache directory with the new disk page address, and the dirty bit is set to 0 (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). Additional information, such as the transaction id(s) of the transaction(s) that modified the buffer, are also kept in the directory. When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk page *only if its dirty bit is 1*.

Another bit, called the **pin-unpin** bit, is also needed—a page in the cache is **pinned** (bit value 1 (one)) if it cannot be written back to disk as yet. For example, the recovery protocol may restrict certain buffer pages from being written back to the disk until the transactions that changed this buffer have committed.

Two main strategies can be employed when flushing a modified buffer back to disk. The first strategy, known as **in-place updating**, writes the buffer to the *same original disk location*, thus overwriting the old value of any changed data items on disk.<sup>2</sup> Hence, a single copy of each database disk block is maintained. The second strategy, known as **shadowing**, writes an updated buffer at a different disk location, so multiple versions of data items can be maintained, but this approach is not typically used in practice.

In general, the old value of the data item before updating is called the **before image (BFIM)**, and the new value after updating is called the **after image (AFIM)**. If shadowing is used, both the BFIM and the AFIM can be kept on disk; hence, it is not strictly necessary to maintain a log for recovering. We briefly discuss recovery based on shadowing in Section 22.4.

### 22.1.3 Write-Ahead Logging, Steal/No-Steal, and Force/No-Force

When in-place updating is used, it is necessary to use a log for recovery (see Section 21.2.2). In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk. This process is generally known as **write-ahead logging** and is necessary so we can UNDO the operation if this is required during recovery. Before we can describe a protocol for write-ahead logging, we need to distinguish between two types of log entry information included for a write command: the information needed for UNDO

---

<sup>2</sup>In-place updating is used in most systems in practice.

and the information needed for REDO. A **REDO-type log entry** includes the **new value** (AFIM) of the item written by the operation since this is needed to *redo* the effect of the operation from the log (by setting the item value in the database on disk to its AFIM). The **UNDO-type log entries** include the **old value** (BFIM) of the item since this is needed to *undo* the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both BFIM and AFIM are recorded into a single log entry. Additionally, when cascading rollback (see Section 22.1.5) is possible, *read\_item* entries in the log are considered to be UNDO-type entries.

As mentioned, the DBMS cache holds the cached database disk blocks in main memory buffers. The DBMS cache includes not only *data file blocks*, but also *index file blocks* and *log file blocks* from the disk. When a log record is written, it is stored in the current log buffer in the DBMS cache. The log is simply a sequential (append-only) disk file, and the DBMS cache may contain several log blocks in main memory buffers (typically, the last  $n$  log blocks of the log file). When an update to a data block—stored in the DBMS cache—is made, an associated log record is written to the last log buffer in the DBMS cache. With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update *must first be written to disk* before the data block itself can be written back to disk from its main memory buffer.

Standard DBMS recovery terminology includes the terms **steal/no-steal** and **force/no-force**, which specify the rules that govern *when* a page from the database cache can be written to disk:

1. If a cache buffer page updated by a transaction *cannot* be written to disk before the transaction commits, the recovery method is called a **no-steal approach**. The pin-unpin bit will be set to 1 (pin) to indicate that a cache buffer cannot be written back to disk. On the other hand, if the recovery protocol allows writing an updated buffer *before* the transaction commits, it is called **steal**. Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed. The *no-steal rule* means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.
2. If all pages updated by a transaction are immediately written to disk *before* the transaction commits, the recovery approach is called a **force approach**. Otherwise, it is called **no-force**. The *force rule* means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.

The deferred update (NO-UNDO) recovery scheme discussed in Section 22.2 follows a *no-steal* approach. However, typical database systems employ a *steal/no-force* (UNDO/REDO) strategy. The *advantage of steal* is that it avoids the need for a very large buffer space to store all updated pages in memory. The *advantage of no-force* is that an updated page of a committed transaction may still be in the buffer when

another transaction needs to update it, thus eliminating the I/O cost to write that page multiple times to disk and possibly having to read it again from disk. This may provide a substantial saving in the number of disk I/O operations when a specific page is updated heavily by multiple transactions.

To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the log on disk before changes are applied to the database. For example, consider the following **write-ahead logging (WAL)** protocol for a recovery algorithm that requires both UNDO and REDO:

1. The before image of an item cannot be overwritten by its after image in the database on disk until all UNDO-type log entries for the updating transaction—up to this point—have been force-written to disk.
2. The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force-written to disk.

To facilitate the recovery process, the DBMS recovery subsystem may need to maintain a number of lists related to the transactions being processed in the system. These include a list for **active transactions** that have started but not committed as yet, and they may also include lists of all **committed** and **aborted transactions** since the last checkpoint (see the next section). Maintaining these lists makes the recovery process more efficient.

#### 22.1.4 Checkpoints in the System Log and Fuzzy Checkpointing

Another type of entry in the log is called a **checkpoint**.<sup>3</sup> A [checkpoint, *list of active transactions*] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their [commit, *T*] entries in the log before a [checkpoint] entry do not need to have their WRITE operations *redone* in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing. As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.

The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time—say, every *m* minutes—or in the number *t* of committed transactions since the last checkpoint, where the values of *m* or *t* are system parameters. Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.
2. Force-write all main memory buffers that have been modified to disk.

---

<sup>3</sup>The term *checkpoint* has been used to describe more restrictive situations in some systems, such as DB2. It has also been used in the literature to describe entirely different concepts.

3. Write a [checkpoint] record to the log, and force-write the log to disk.
4. Resume executing transactions.

As a consequence of step 2, a checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction. This can facilitate undoing transaction operations in the event that a transaction must be rolled back.

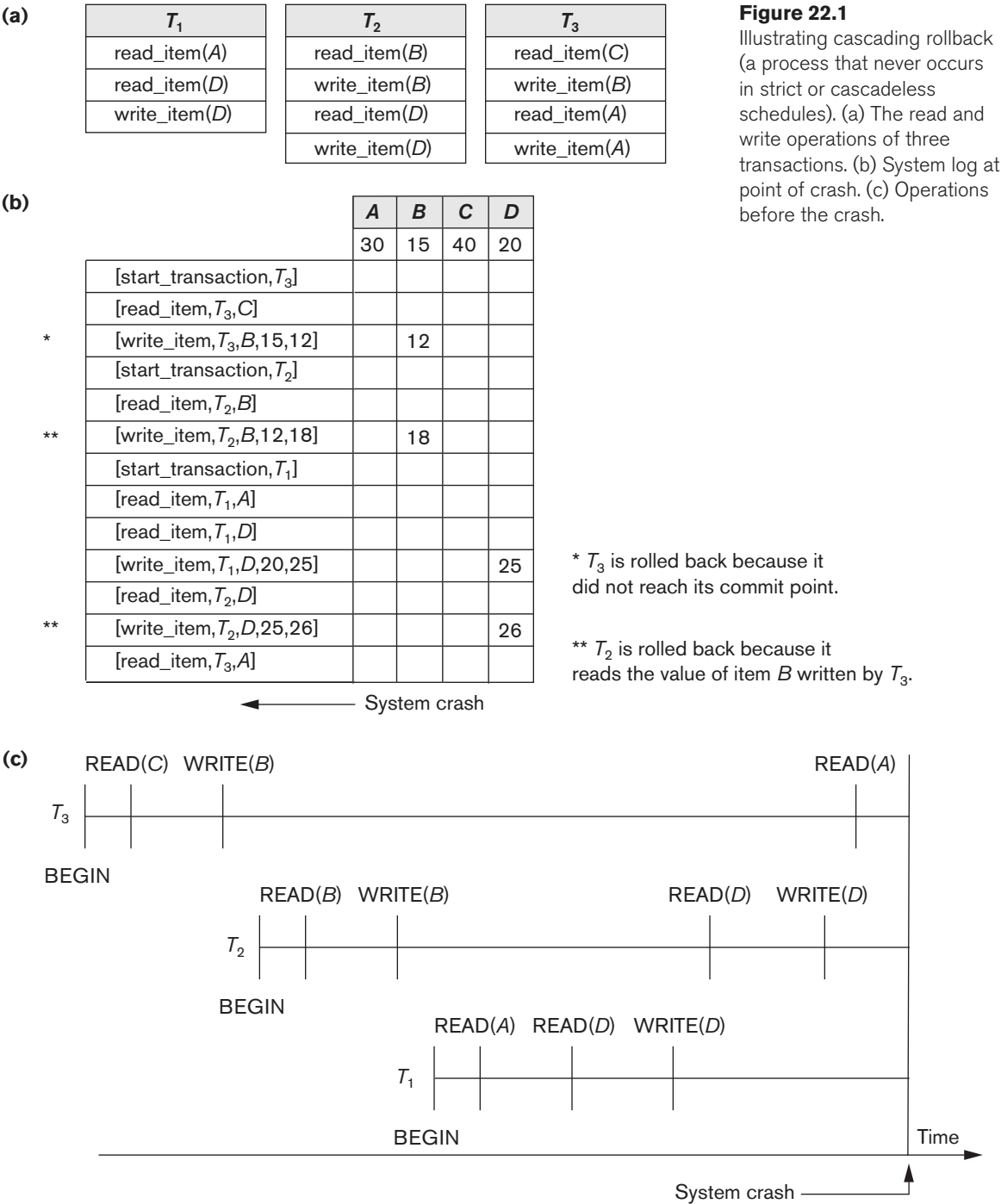
The time needed to force-write all modified memory buffers may delay transaction processing because of step 1, which is not acceptable in practice. To overcome this, it is common to use a technique called **fuzzy checkpointing**. In this technique, the system can resume transaction processing after a [begin\_checkpoint] record is written to the log without having to wait for step 2 to finish. When step 2 is completed, an [end\_checkpoint, ... ] record is written in the log with the relevant information collected during checkpointing. However, until step 2 is completed, the previous checkpoint record should remain valid. To accomplish this, the system maintains a file on disk that contains a pointer to the valid checkpoint, which continues to point to the previous checkpoint record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

### 22.1.5 Transaction Rollback and Cascading Rollback

If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to **roll back** the transaction. If any data item values have been changed by the transaction and written to the database on disk, they must be restored to their previous values (BFIMs). The undo-type log entries are used to restore the old values of data items that must be rolled back.

If a transaction  $T$  is rolled back, any transaction  $S$  that has, in the interim, read the value of some data item  $X$  written by  $T$  must also be rolled back. Similarly, once  $S$  is rolled back, any transaction  $R$  that has read the value of some data item  $Y$  written by  $S$  must also be rolled back; and so on. This phenomenon is called **cascading rollback**, and it can occur when the recovery protocol ensures *recoverable* schedules but does not ensure *strict* or *cascadeless* schedules (see Section 20.4.2). Understandably, cascading rollback can be complex and time-consuming. That is why almost all recovery mechanisms are designed so that cascading rollback *is never required*.

Figure 22.1 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 22.1(a). Figure 22.1(b) shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of data items  $A$ ,  $B$ ,  $C$ , and  $D$ , which are used by the transactions, are shown to the right of the system log entries. We assume that the original item values, shown in the first line, are  $A = 30$ ,  $B = 15$ ,  $C = 40$ , and  $D = 20$ . At the point of system failure, transaction  $T_3$  has not reached its conclusion and must be rolled back. The WRITE operations of  $T_3$ , marked by a single \* in Figure 22.1(b), are the  $T_3$  operations that are undone during transaction rollback. Figure 22.1(c) graphically shows the operations of the different transactions along the time axis.



**Figure 22.1**  
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

\*  $T_3$  is rolled back because it did not reach its commit point.

\*\*  $T_2$  is rolled back because it reads the value of item  $B$  written by  $T_3$ .



We must now check for cascading rollback. From Figure 22.1(c), we see that transaction  $T_2$  reads the value of item  $B$  that was written by transaction  $T_3$ ; this can also be determined by examining the log. Because  $T_3$  is rolled back,  $T_2$  must now be rolled back, too. The WRITE operations of  $T_2$ , marked by \*\* in the log, are the ones that are undone. Note that only write\_item operations need to be undone during transaction rollback; read\_item operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.

In practice, cascading rollback of transactions is *never* required because practical recovery methods *guarantee cascadeless or strict* schedules. Hence, there is also no need to record any read\_item operations in the log because these are needed only for determining cascading rollback.

### 22.1.6 Transaction Actions That Do Not Affect the Database

In general, a transaction will have actions that do *not* affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action that is based on these reports and that affects the database. Hence, such reports should be generated only *after the transaction reaches its commit point*. A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

## 22.2 NO-UNDO/REDO Recovery Based on Deferred Update

The idea behind deferred update is to defer or postpone any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.<sup>4</sup>

During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database. If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way. Therefore, only **REDO-type log entries** are needed in the log, which include the **new value** (AFIM) of the item written by a write operation. The **UNDO-type log entries** are not needed since no undoing of operations will be required during recovery. Although this may simplify the recovery process, it cannot be used in practice unless transactions are short and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held

---

<sup>4</sup>Hence deferred update can generally be characterized as a *no-steal approach*.



in the cache buffers until the commit point, so many cache buffers will be *pinned* and cannot be replaced.

We can state a typical deferred update protocol as follows:

1. A transaction cannot change the database on disk until it reaches its commit point; hence all buffers that have been changed by the transaction must be pinned until the transaction commits (this corresponds to a *no-steal policy*).
2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log *and* the log buffer is force-written to disk.

Notice that step 2 of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations. REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk. In this case, the transaction operations are redone from the log entries during recovery.

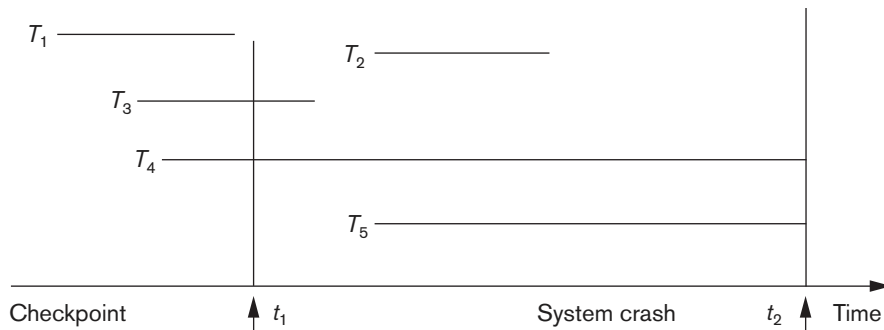
For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated. Consider a system in which concurrency control uses strict two-phase locking, so the locks on written items remain in effect *until the transaction reaches its commit point*. After that, the locks can be released. This ensures strict and serializable schedules. Assuming that [checkpoint] entries are included in the log, a possible recovery algorithm for this case, which we call RDU\_M (Recovery using Deferred Update in a Multiuser environment), is given next.

**Procedure RDU\_M (NO-UNDO/REDO with checkpoints).** Use two lists of transactions maintained by the system: the committed transactions  $T$  since the last checkpoint (**commit list**), and the active transactions  $T'$  (**active list**). REDO all the WRITE operations of the committed transactions from the log, *in the order in which they were written into the log*. The transactions that are active and did not commit are effectively canceled and must be resubmitted.

The REDO procedure is defined as follows:

**Procedure REDO (WRITE\_OP).** Redoing a write\_item operation WRITE\_OP consists of examining its log entry [write\_item,  $T$ ,  $X$ , new\_value] and setting the value of item  $X$  in the database to new\_value, which is the after image (AFIM).

Figure 22.2 illustrates a timeline for a possible schedule of executing transactions. When the checkpoint was taken at time  $t_1$ , transaction  $T_1$  had committed, whereas transactions  $T_3$  and  $T_4$  had not. Before the system crash at time  $t_2$ ,  $T_3$  and  $T_2$  were committed but not  $T_4$  and  $T_5$ . According to the RDU\_M method, there is no need to redo the write\_item operations of transaction  $T_1$ —or any transactions committed before the last checkpoint time  $t_1$ . The write\_item operations of  $T_2$  and  $T_3$  must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions  $T_4$  and  $T_5$  are ignored: They are effectively canceled or rolled back because none of their write\_item operations were recorded in the database on disk under the deferred update protocol (no-steal policy).

**Figure 22.2**

An example of a recovery timeline to illustrate the effect of checkpointing.

We can make the NO-UNDO/REDO recovery algorithm *more efficient* by noting that, if a data item  $X$  has been updated—as indicated in the log entries—more than once by committed transactions since the last checkpoint, it is only necessary to REDO *the last update of  $X$*  from the log during recovery because the other updates would be overwritten by this last REDO. In this case, we start from *the end of the log*; then, whenever an item is redone, it is added to a list of redone items. Before REDO is applied to an item, the list is checked; if the item appears on the list, it is not redone again, since its latest value has already been recovered.

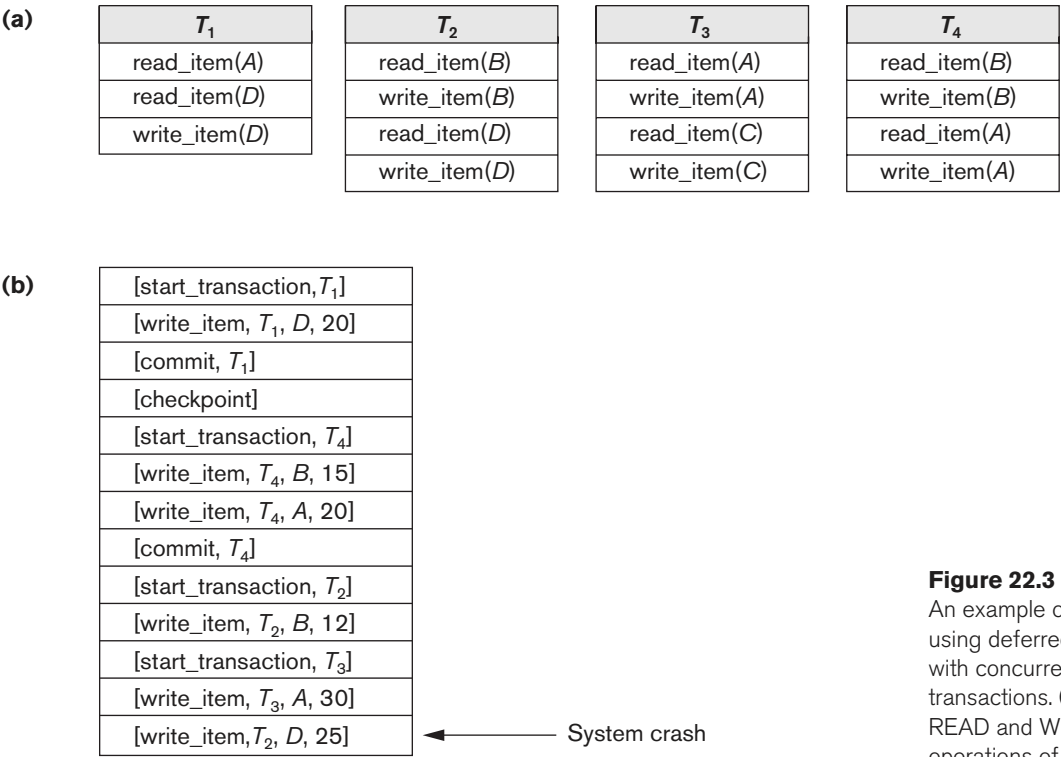
If a transaction is aborted for any reason (say, by the deadlock detection method), it is simply resubmitted, since it has not changed the database on disk. A drawback of the method described here is that it limits the concurrent execution of transactions because *all write-locked items remain locked until the transaction reaches its commit point*. Additionally, it may require excessive buffer space to hold all updated items until the transactions commit. The method's main benefit is that transaction operations *never need to be undone*, for two reasons:

1. A transaction does not record any changes in the database on disk until after it reaches its commit point—that is, until it completes its execution successfully. Hence, a transaction is never rolled back because of failure during transaction execution.
2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur.

Figure 22.3 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method just described.

## 22.3 Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the database on disk can be updated *immediately*, without any need to wait for the transaction to reach its commit point. Notice that it is *not a requirement* that every update be



**Figure 22.3**  
An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

$T_2$  and  $T_3$  are ignored because they did not reach their commit points.  
 $T_4$  is redone because its commit point is after the last system checkpoint.

applied immediately to disk; it is just possible that some updates are applied to disk *before the transaction commits*.

Provisions must be made for *undoing* the effect of update operations that have been applied to the database by a *failed transaction*. This is accomplished by rolling back the transaction and undoing the effect of the transaction's write\_item operations. Therefore, the **UNDO-type log entries**, which include the **old value** (BFIM) of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a **steal strategy** for deciding when updated main memory buffers can be written back to disk (see Section 22.1.3).

Theoretically, we can distinguish two main categories of immediate update algorithms.

1. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits*, there is never a need to REDO any operations of committed transactions. This is called the **UNDO/NO-REDO recovery algorithm**. In this method, all updates by a transaction must be recorded on disk *before the transaction commits*, so that REDO is never needed. Hence, this method must utilize the **steal/force**

**strategy** for deciding when updated main memory buffers are written back to disk (see Section 22.1.3).

2. If the transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the **UNDO/REDO recovery algorithm**. In this case, the **steal/no-force strategy** is applied (see Section 22.1.3). This is also the most complex technique, but the most commonly used in practice. We will outline an UNDO/REDO recovery algorithm and leave it as an exercise for the reader to develop the UNDO/NO-REDO variation. In Section 22.5, we describe a more practical approach known as the ARIES recovery technique.

When concurrent execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure RIU\_M (Recovery using Immediate Updates for a Multiuser environment) outlines a recovery algorithm for concurrent transactions with immediate update (UNDO/REDO recovery). Assume that the log includes checkpoints and that the concurrency control protocol produces *strict schedules*—as, for example, the strict two-phase locking protocol does. Recall that a strict schedule does not allow a transaction to read or write an item unless the transaction that wrote the item has committed. However, deadlocks can occur in strict two-phase locking, thus requiring abort and UNDO of transactions. For a strict schedule, UNDO of an operation requires changing the item back to its old value (BFIM).

**Procedure RIU\_M (UNDO/REDO with checkpoints).**

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.
2. Undo all the `write_item` operations of the *active* (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3. Redo all the `write_item` operations of the *committed* transactions from the log, in the order in which they were written into the log, using the REDO procedure defined earlier.

The UNDO procedure is defined as follows:

**Procedure UNDO (WRITE\_OP).** Undoing a `write_item` operation `write_op` consists of examining its log entry [`write_item`, `T`, `X`, `old_value`, `new_value`] and setting the value of item `X` in the database to `old_value`, which is the before image (BFIM). Undoing a number of `write_item` operations from one or more transactions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

As we discussed for the **NO-UNDO/REDO** procedure, step 3 is more efficiently done by starting from the *end of the log* and redoing only *the last update of each item X*. Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2 so that an item can be undone at most once during recovery. In this case, the earliest UNDO is applied first by scanning the log in the forward direction (starting from

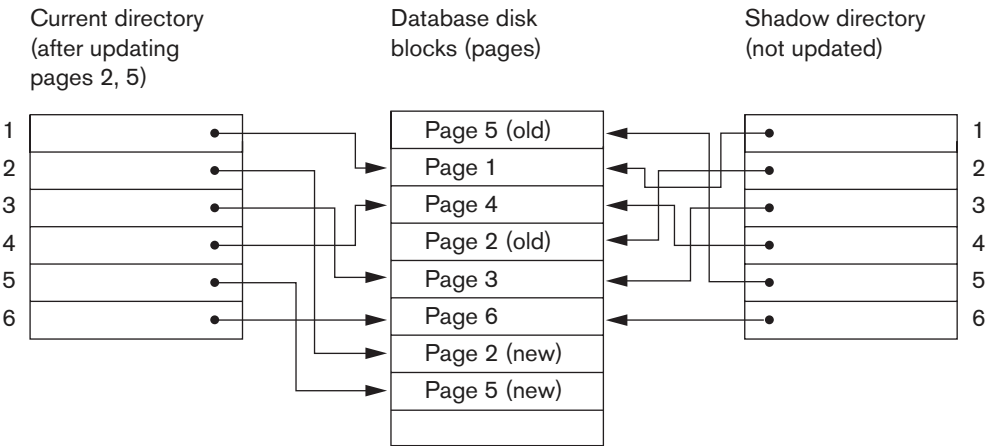
the beginning of the log). Whenever an item is undone, it is added to a list of undone items and is not undone again.

## 22.4 Shadow Paging

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method. Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)—say,  $n$ —for recovery purposes. A **directory** with  $n$  entries<sup>5</sup> is constructed, where the  $i$ th entry points to the  $i$ th database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory**. The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is *never* modified. When a write\_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten*. Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure 22.4 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

**Figure 22.4**  
An example of shadow paging.



<sup>5</sup>The directory is similar to the page table maintained by the operating system for each process.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant. A further complication is how to handle **garbage collection** when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits. Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

## 22.5 The ARIES Recovery Algorithm

We now describe the ARIES algorithm as an example of a recovery algorithm used in database systems. It is used in many relational database-related products of IBM. ARIES uses a steal/no-force approach for writing, and it is based on three concepts: write-ahead logging, repeating history during redo, and logging changes during undo. We discussed write-ahead logging in Section 22.1.3. The second concept, **repeating history**, means that ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state *when the crash occurred*. Transactions that were uncommitted at the time of the crash (active transactions) are undone. The third concept, **logging during undo**, will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES recovery procedure consists of three main steps: analysis, REDO, and UNDO. The **analysis step** identifies the dirty (updated) pages in the buffer<sup>6</sup> and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined. The **REDO phase** actually reapplies updates from the log to the database. Generally, the REDO operation is applied only to committed transactions. However, this is not the case in ARIES.

---

<sup>6</sup>The actual buffers may be lost during a crash, since they are in main memory. Additional tables stored in the log during checkpointing (Dirty Page Table, Transaction Table) allow ARIES to identify this information (as discussed later in this section).

Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. Additionally, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied to the database and therefore does not need to be reapplied. Thus, *only the necessary REDO operations* are applied during recovery. Finally, during the **UNDO phase**, the log is scanned backward and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. Additionally, checkpointing is used. These tables are maintained by the transaction manager and written to the log during checkpointing.

In ARIES, every log record has an associated **log sequence number (LSN)** that is monotonically increasing and indicates the address of the log record on disk. Each LSN corresponds to a *specific change* (action) of some transaction. Also, each data page will store the LSN of the *latest log record corresponding to a change for that page*. A log record is written for any of the following actions: updating a page (write), committing a transaction (commit), aborting a transaction (abort), undoing an update (undo), and ending a transaction (end). The need for including the first three actions in the log has been discussed, but the last two need some explanation. When an update is undone, a *compensation log record* is written in the log so that the undo does not have to be repeated. When a transaction ends, whether by committing or aborting, an *end log record* is written.

Common fields in all log records include the previous LSN for that transaction, the transaction ID, and the type of log record. The previous LSN is important because it links the log records (in reverse order) for each transaction. For an update (write) action, additional fields in the log record include the page ID for the page that contains the item, the length of the updated item, its offset from the beginning of the page, the before image of the item, and its after image.

In addition to the log, two tables are needed for efficient recovery: the **Transaction Table** and the **Dirty Page Table**, which are maintained by the transaction manager. When a crash occurs, these tables are rebuilt in the analysis phase of recovery. The Transaction Table contains an entry for *each active transaction*, with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction. The Dirty Page Table contains an entry for each dirty page in the DBMS cache, which includes the page ID and the LSN corresponding to the earliest update to that page.

**Checkpointing** in ARIES consists of the following: writing a `begin_checkpoint` record to the log, writing an `end_checkpoint` record to the log, and writing the LSN of the `begin_checkpoint` record to a special file. This special file is accessed during recovery to locate the last checkpoint information. With the `end_checkpoint` record, the contents of both the Transaction Table and Dirty Page Table are appended to the end of the log. To reduce the cost, **fuzzy checkpointing** is used so that the DBMS can continue to execute transactions during checkpointing (see Section 22.1.4). Additionally, the contents of the DBMS cache do not have to be flushed



to disk during checkpoint, since the Transaction Table and Dirty Page Table—which are appended to the log on disk—contain the information needed for recovery. Note that if a crash occurs during checkpointing, the special file will refer to the previous checkpoint, which would be used for recovery.

After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file. The **analysis phase** starts at the `begin_checkpoint` record and proceeds to the end of the log. When the `end_checkpoint` record is encountered, the Transaction Table and Dirty Page Table are accessed (recall that these tables were written in the log during checkpointing). During analysis, the log records being analyzed may cause modifications to these two tables. For instance, if an end log record was encountered for a transaction  $T$  in the Transaction Table, then the entry for  $T$  is deleted from that table. If some other type of log record is encountered for a transaction  $T'$ , then an entry for  $T'$  is inserted into the Transaction Table, if not already present, and the last LSN field is modified. If the log record corresponds to a change for page  $P$ , then an entry would be made for page  $P$  (if not present in the table) and the associated LSN field would be modified. When the analysis phase is complete, the necessary information for REDO and UNDO has been compiled in the tables.

The **REDO phase** follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages *have already been applied to the database on disk*. It can determine this by finding the smallest LSN,  $M$ , of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to an LSN  $< M$ , for redoable transactions, must have already been propagated to disk or already been overwritten in the buffer; otherwise, those dirty pages with that LSN would be in the buffer (and the Dirty Page Table). So, REDO starts at the log record with LSN =  $M$  and scans forward to the end of the log.

For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. For example, if a change recorded in the log pertains to page  $P$  that is not in the Dirty Page Table, then this change is already on disk and does not need to be reapplied. Or, if a change recorded in the log (with LSN =  $N$ , say) pertains to page  $P$  and the Dirty Page Table contains an entry for  $P$  with LSN greater than  $N$ , then the change is already present. If neither of these two conditions holds, page  $P$  is read from disk and the LSN stored on that page,  $LSN(P)$ , is compared with  $N$ . If  $N < LSN(P)$ , then the change has been applied and the page does not need to be rewritten to disk.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions—called the `undo_set`—has been identified in the Transaction Table during the analysis phase. Now, the **UNDO phase** proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of transactions in the `undo_set` has been undone. When this is completed, the recovery process is finished and normal processing can begin again.



(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	$T_1$	update	C	...
2	0	$T_2$	update	B	...
3	1	$T_1$	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	$T_3$	update	A	...
7	2	$T_2$	update	C	...
8	7	$T_2$	commit		...

(b)	TRANSACTION TABLE			DIRTY PAGE TABLE	
	Transaction_id	Last_lsn	Status	Page_id	Lsn
	$T_1$	3	commit	C	1
	$T_2$	2	in progress	B	2

(c)	TRANSACTION TABLE			DIRTY PAGE TABLE	
	Transaction_id	Last_lsn	Status	Page_id	Lsn
	$T_1$	3	commit	C	7
	$T_2$	8	commit	B	2
	$T_3$	6	in progress	A	6

**Figure 22.5**

An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.

Consider the recovery example shown in Figure 22.5. There are three transactions:  $T_1$ ,  $T_2$ , and  $T_3$ .  $T_1$  updates page C,  $T_2$  updates pages B and C, and  $T_3$  updates page A. Figure 22.5(a) shows the partial contents of the log, and Figure 22.5(b) shows the contents of the Transaction Table and Dirty Page Table. Now, suppose that a crash occurs at this point. Since a checkpoint has occurred, the address of the associated begin\_checkpoint record is retrieved, which is location 4. The analysis phase starts from location 4 until it reaches the end. The end\_checkpoint record contains the Transaction Table and Dirty Page Table in Figure 22.5(b), and the analysis phase will further reconstruct these tables. When the analysis phase encounters log record 6, a new entry for transaction  $T_3$  is made in the Transaction Table and a new entry for page A is made in the Dirty Page Table. After log record 8 is analyzed, the status of transaction  $T_2$  is changed to committed in the Transaction Table. Figure 22.5(c) shows the two tables after the analysis phase.

For the REDO phase, the smallest LSN in the Dirty Page Table is 1. Hence the REDO will start at log record 1 and proceed with the REDO of updates. The LSNs {1, 2, 6, 7} corresponding to the updates for pages C, B, A, and C, respectively, are not less than the LSNs of those pages (as shown in the Dirty Page Table). So those data pages will be read again and the updates reapplied from the log (assuming the actual LSNs stored on those data pages are less than the corresponding log entry). At this point, the REDO phase is finished and the UNDO phase starts. From the Transaction Table (Figure 22.5(c)), UNDO is applied only to the active transaction  $T_3$ . The UNDO phase starts at log entry 6 (the last update for  $T_3$ ) and proceeds backward in the log. The backward chain of updates for transaction  $T_3$  (only log record 6 in this example) is followed and undone.

## 22.6 Recovery in Multidatabase Systems

So far, we have implicitly assumed that a transaction accesses a single database. In some cases, a single transaction, called a **multidatabase transaction**, may require access to multiple databases. These databases may even be stored on different types of DBMSs; for example, some DBMSs may be relational, whereas others are object-oriented, hierarchical, or network DBMSs. In such a case, each DBMS involved in the multidatabase transaction may have its own recovery technique and transaction manager separate from those of the other DBMSs. This situation is somewhat similar to the case of a distributed database management system (see Chapter 23), where parts of the database reside at different sites that are connected by a communication network.

To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism. A **global recovery manager**, or **coordinator**, is needed to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables). The coordinator usually follows a protocol called the **two-phase commit protocol**, whose two phases can be stated as follows:

- **Phase 1.** When all participating databases signal the coordinator that the part of the multidatabase transaction involving each has concluded, the coordinator sends a message *prepare for commit* to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records and needed information for local recovery to disk and then send a *ready to commit* or *OK* signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a *cannot commit* or *not OK* signal to the coordinator. If the coordinator does not receive a reply from the database within a certain time out interval, it assumes a *not OK* response.
- **Phase 2.** If *all* participating databases reply *OK*, and the coordinator's vote is also *OK*, the transaction is successful, and the coordinator sends a *commit* signal for the transaction to the participating databases. Because all the local effects of the transaction and information needed for local recovery have

been recorded in the logs of the participating databases, local recovery from failure is now possible. Each participating database completes transaction commit by writing a [commit] entry for the transaction in the log and permanently updating the database if needed. Conversely, if one or more of the participating databases or the coordinator have a *not OK* response, the transaction has failed, and the coordinator sends a message to *roll back* or *UNDO* the local effect of the transaction to each participating database. This is done by undoing the local transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants—or the coordinator—fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before phase 1 usually requires the transaction to be rolled back, whereas a failure during phase 2 means that a successful transaction can recover and commit.

## 22.7 Database Backup and Recovery from Catastrophic Failures

So far, all the techniques we have discussed apply to noncatastrophic failures. A key assumption has been that the system log is maintained on the disk and is not lost as a result of the failure. Similarly, the shadow directory must be stored on disk to allow recovery when shadow paging is used. The recovery techniques we have discussed use the entries in the system log or the shadow directory to recover from failure by bringing the database back to a consistent state.

The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes. The main technique used to handle such crashes is a **database backup**, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations. Subterranean storage vaults have been used to protect such data from flood, storm, earthquake, or fire damage. Events like the 9/11 terrorist attack in New York (in 2001) and the Katrina hurricane disaster in New Orleans (in 2005) have created a greater awareness of *disaster recovery of critical databases*.

To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape. The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. Therefore, users do not lose all transactions they have performed since the last database backup. All committed transactions recorded in the portion of the system log that has been backed up to tape can have their effect on the database

redone. A new log is started after each database backup. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been recorded in the backed-up copies of the system log are reconstructed.

## 22.8 Summary

In this chapter, we discussed the techniques for recovery from transaction failures. The main goal of recovery is to ensure the atomicity property of a transaction. If a transaction fails before completing its execution, the recovery mechanism has to make sure that the transaction has no lasting effects on the database. First in Section 22.1 we gave an informal outline for a recovery process, and then we discussed system concepts for recovery. These included a discussion of caching, in-place updating versus shadowing, before and after images of a data item, UNDO versus REDO recovery operations, steal/no-steal and force/no-force policies, system checkpointing, and the write-ahead logging protocol.

Next we discussed two different approaches to recovery: deferred update (Section 22.2) and immediate update (Section 22.3). Deferred update techniques postpone any actual updating of the database on disk until a transaction reaches its commit point. The transaction force-writes the log to disk before recording the updates in the database. This approach, when used with certain concurrency control methods, is designed never to require transaction rollback, and recovery simply consists of redoing the operations of transactions committed after the last checkpoint from the log. The disadvantage is that too much buffer space may be needed, since updates are kept in the buffers and are not applied to disk until a transaction commits. Deferred update can lead to a recovery algorithm known as NO-UNDO/REDO. Immediate update techniques may apply changes to the database on disk before the transaction reaches a successful conclusion. Any changes applied to the database must first be recorded in the log and force-written to disk so that these operations can be undone if necessary. We also gave an overview of a recovery algorithm for immediate update known as UNDO/REDO. Another algorithm, known as UNDO/NO-REDO, can also be developed for immediate update if all transaction actions are recorded in the database before commit.

We discussed the shadow paging technique for recovery in Section 22.4, which keeps track of old database pages by using a shadow directory. This technique, which is classified as NO-UNDO/NO-REDO, does not require a log in single-user systems but still needs the log for multiuser systems. We also presented ARIES in Section 22.5, which is a specific recovery scheme used in many of IBM's relational database products. Then in Section 22.6 we discussed the two-phase commit protocol, which is used for recovery from failures involving multidatabase transactions. Finally, we discussed recovery from catastrophic failures in Section 22.7, which is typically done by backing up the database and the log to tape. The log can be backed up more frequently than the database, and the backup log can be used to redo operations starting from the last database backup.

## Review Questions

- 22.1. Discuss the different types of transaction failures. What is meant by *catastrophic failure*?
- 22.2. Discuss the actions taken by the `read_item` and `write_item` operations on a database.
- 22.3. What is the system log used for? What are the typical kinds of entries in a system log? What are checkpoints, and why are they important? What are transaction commit points, and why are they important?
- 22.4. How are buffering and caching techniques used by the recovery subsystem?
- 22.5. What are the before image (BFIM) and after image (AFIM) of a data item? What is the difference between in-place updating and shadowing, with respect to their handling of BFIM and AFIM?
- 22.6. What are UNDO-type and REDO-type log entries?
- 22.7. Describe the write-ahead logging protocol.
- 22.8. Identify three typical lists of transactions that are maintained by the recovery subsystem.
- 22.9. What is meant by *transaction rollback*? What is meant by *cascading rollback*? Why do practical recovery methods use protocols that do not permit cascading rollback? Which recovery techniques do not require any rollback?
- 22.10. Discuss the UNDO and REDO operations and the recovery techniques that use each.
- 22.11. Discuss the deferred update technique of recovery. What are the advantages and disadvantages of this technique? Why is it called the NO-UNDO/REDO method?
- 22.12. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by a transaction?
- 22.13. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update?
- 22.14. What is the difference between the UNDO/REDO and the UNDO/NO-REDO algorithms for recovery with immediate update? Develop the outline for an UNDO/NO-REDO algorithm.
- 22.15. Describe the shadow paging recovery technique. Under what circumstances does it not require a log?
- 22.16. Describe the three phases of the ARIES recovery method.
- 22.17. What are log sequence numbers (LSNs) in ARIES? How are they used? What information do the Dirty Page Table and Transaction Table contain? Describe how fuzzy checkpointing is used in ARIES.

- 22.18.** What do the terms *steal/no-steal* and *force/no-force* mean with regard to buffer management for transaction processing?
- 22.19.** Describe the two-phase commit protocol for multidatabase transactions.
- 22.20.** Discuss how disaster recovery from catastrophic failures is handled.

## Exercises

- 22.21.** Suppose that the system crashes before the  $[\text{read\_item}, T_3, A]$  entry is written to the log in Figure 22.1(b). Will that make any difference in the recovery process?
- 22.22.** Suppose that the system crashes before the  $[\text{write\_item}, T_2, D, 25, 26]$  entry is written to the log in Figure 22.1(b). Will that make any difference in the recovery process?
- 22.23.** Figure 22.6 shows the log corresponding to a particular schedule at the point of a system crash for four transactions  $T_1$ ,  $T_2$ ,  $T_3$ , and  $T_4$ . Suppose that we use the *immediate update protocol* with checkpointing. Describe the recovery process from the system crash. Specify which transactions are rolled back, which operations in the log are redone and which (if any) are undone, and whether any cascading rollback takes place.

$[\text{start\_transaction}, T_1]$
$[\text{read\_item}, T_1, A]$
$[\text{read\_item}, T_1, D]$
$[\text{write\_item}, T_1, D, 20, 25]$
$[\text{commit}, T_1]$
$[\text{checkpoint}]$
$[\text{start\_transaction}, T_2]$
$[\text{read\_item}, T_2, B]$
$[\text{write\_item}, T_2, B, 12, 18]$
$[\text{start\_transaction}, T_4]$
$[\text{read\_item}, T_4, D]$
$[\text{write\_item}, T_4, D, 25, 15]$
$[\text{start\_transaction}, T_3]$
$[\text{write\_item}, T_3, C, 30, 40]$
$[\text{read\_item}, T_4, A]$
$[\text{write\_item}, T_4, A, 30, 20]$
$[\text{commit}, T_4]$
$[\text{read\_item}, T_2, D]$
$[\text{write\_item}, T_2, D, 15, 25]$

← System crash

**Figure 22.6**

A sample schedule and its corresponding log.

- 22.24.** Suppose that we use the deferred update protocol for the example in Figure 22.6. Show how the log would be different in the case of deferred update by removing the unnecessary log entries; then describe the recovery process, using your modified log. Assume that only REDO operations are applied, and specify which operations in the log are redone and which are ignored.
- 22.25.** How does checkpointing in ARIES differ from checkpointing as described in Section 22.1.4?
- 22.26.** How are log sequence numbers used by ARIES to reduce the amount of REDO work needed for recovery? Illustrate with an example using the information shown in Figure 22.5. You can make your own assumptions as to when a page is written to disk.
- 22.27.** What implications would a no-steal/force buffer management policy have on checkpointing and recovery?

*Choose the correct answer for each of the following multiple-choice questions:*

- 22.28.** Incremental logging with deferred updates implies that the recovery system must
- store the old value of the updated item in the log
  - store the new value of the updated item in the log
  - store both the old and new value of the updated item in the log
  - store only the Begin Transaction and Commit Transaction records in the log
- 22.29.** The write-ahead logging (WAL) protocol simply means that
- writing of a data item should be done ahead of any logging operation
  - the log record for an operation should be written before the actual data is written
  - all log records should be written before a new transaction begins execution
  - the log never needs to be written to disk
- 22.30.** In case of transaction failure under a deferred update incremental logging scheme, which of the following will be needed?
- an undo operation
  - a redo operation
  - an undo and redo operation
  - none of the above
- 22.31.** For incremental logging with immediate updates, a log record for a transaction would contain
- a transaction name, a data item name, and the old and new value of the item
  - a transaction name, a data item name, and the old value of the item
  - a transaction name, a data item name, and the new value of the item
  - a transaction name and a data item name

- 22.32.** For correct behavior during recovery, undo and redo operations must be
- commutative
  - associative
  - idempotent
  - distributive
- 22.33.** When a failure occurs, the log is consulted and each operation is either undone or redone. This is a problem because
- searching the entire log is time consuming
  - many redos are unnecessary
  - both (a) and (b)
  - none of the above
- 22.34.** Using a log-based recovery scheme might improve performance as well as provide a recovery mechanism by
- writing the log records to disk when each transaction commits
  - writing the appropriate log records to disk during the transaction's execution
  - waiting to write the log records until multiple transactions commit and writing them as a batch
  - never writing the log records to disk
- 22.35.** There is a possibility of a cascading rollback when
- a transaction writes items that have been written only by a committed transaction
  - a transaction writes an item that is previously written by an uncommitted transaction
  - a transaction reads an item that is previously written by an uncommitted transaction
  - both (b) and (c)
- 22.36.** To cope with media (disk) failures, it is necessary
- for the DBMS to only execute transactions in a single user environment
  - to keep a redundant copy of the database
  - to never abort a transaction
  - all of the above
- 22.37.** If the shadowing approach is used for flushing a data item back to disk, then
- the item is written to disk only after the transaction commits
  - the item is written to a different location on disk
  - the item is written to disk before the transaction commits
  - the item is written to the same disk location from which it was read



## Selected Bibliography

The books by Bernstein et al. (1987) and Papadimitriou (1986) are devoted to the theory and principles of concurrency control and recovery. The book by Gray and Reuter (1993) is an encyclopedic work on concurrency control, recovery, and other transaction-processing issues.

Verhofstad (1978) presents a tutorial and survey of recovery techniques in database systems. Categorizing algorithms based on their UNDO/REDO characteristics is discussed in Haerder and Reuter (1983) and in Bernstein et al. (1983). Gray (1978) discusses recovery, along with other system aspects of implementing operating systems for databases. The shadow paging technique is discussed in Lorie (1977), Verhofstad (1978), and Reuter (1980). Gray et al. (1981) discuss the recovery mechanism in SYSTEM R. Lockemann and Knutsen (1968), Davies (1973), and Bjork (1973) are early papers that discuss recovery. Chandy et al. (1975) discuss transaction roll-back. Lilien and Bhargava (1985) discuss the concept of integrity block and its use to improve the efficiency of recovery.

Recovery using write-ahead logging is analyzed in Jhingran and Khedkar (1992) and is used in the ARIES system (Mohan et al., 1992). More recent work on recovery includes compensating transactions (Korth et al., 1990) and main memory database recovery (Kumar, 1991). The ARIES recovery algorithms (Mohan et al., 1992) have been successful in practice. Franklin et al. (1992) discusses recovery in the EXODUS system. Two books by Kumar and Hsu (1998) and Kumar and Song (1998) discuss recovery in detail and contain descriptions of recovery methods used in a number of existing relational database products. Examples of page replacement strategies that are specific for databases are discussed in Chou and DeWitt (1985) and Pazos et al. (2006).

# part 10

## **Distributed Databases, NOSQL Systems, and Big Data**

This page intentionally left blank

## Distributed Database Concepts

In this chapter, we turn our attention to distributed databases (DDBs), distributed database management systems (DDBMSs), and how the client-server architecture is used as a platform for database application development. Distributed databases bring the advantages of distributed computing to the database domain. A **distributed computing system** consists of a number of processing sites or nodes that are interconnected by a computer network and that cooperate in performing certain assigned tasks. As a general goal, distributed computing systems partition a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. Thus, more computing power is harnessed to solve a complex task, and the autonomous processing nodes can be managed independently while they cooperate to provide the needed functionalities to solve the problem. DDB technology resulted from a merger of two technologies: database technology and distributed systems technology.

Several distributed database prototype systems were developed in the 1980s and 1990s to address the issues of data distribution, data replication, distributed query and transaction processing, distributed database metadata management, and other topics. More recently, many new technologies have emerged that combine distributed and database technologies. These technologies and systems are being developed for dealing with the storage, analysis, and mining of the vast amounts of data that are being produced and collected, and they are referred to generally as **big data technologies**. The origins of big data technologies come from distributed systems and database systems, as well as data mining and machine learning algorithms that can process these vast amounts of data to extract needed knowledge.

In this chapter, we discuss the concepts that are central to data distribution and the management of distributed data. Then in the following two chapters, we give an overview of some of the new technologies that have emerged to manage and process big data. Chapter 24 discusses the new class of database systems known as NOSQL

systems, which focus on providing distributed solutions to manage the vast amounts of data that are needed in applications such as social media, healthcare, and security, to name a few. Chapter 25 introduces the concepts and systems being used for processing and analysis of big data, such as map-reduce and other distributed processing technologies. We also discuss cloud computing concepts in Chapter 25.

Section 23.1 introduces distributed database management and related concepts. Issues of distributed database design, involving fragmenting and sharding of data and distributing it over multiple sites, as well as data replication, are discussed in Section 23.2. Section 23.3 gives an overview of concurrency control and recovery in distributed databases. Sections 23.4 and 23.5 introduce distributed transaction processing and distributed query processing techniques, respectively. Sections 23.6 and 23.7 introduce different types of distributed database systems and their architectures, including federated and multidatabase systems. The problems of heterogeneity and the needs of autonomy in federated database systems are also highlighted. Section 23.8 discusses catalog management schemes in distributed databases. Section 23.9 summarizes the chapter.

For a short introduction to the topic of distributed databases, Sections 23.1 through 23.5 may be covered and the other sections may be omitted.

## 23.1 Distributed Database Concepts

We can define a **distributed database (DDB)** as a collection of multiple logically interrelated databases distributed over a computer network, and a **distributed database management system (DDBMS)** as a software system that manages a distributed database while making the distribution transparent to the user.

### 23.1.1 What Constitutes a DDB

For a database to be called distributed, the following minimum conditions should be satisfied:

- **Connection of database nodes over a computer network.** There are multiple computers, called **sites** or **nodes**. These sites must be connected by an underlying **network** to transmit data and commands among sites.
- **Logical interrelation of the connected databases.** It is essential that the information in the various database nodes be logically related.
- **Possible absence of homogeneity among connected nodes.** It is not necessary that all nodes be identical in terms of data, hardware, and software.

The sites may all be located in physical proximity—say, within the same building or a group of adjacent buildings—and connected via a **local area network**, or they may be geographically distributed over large distances and connected via a **long-haul** or **wide area network**. Local area networks typically use wireless hubs or cables, whereas long-haul networks use telephone lines, cables, wireless communication infrastructures, or satellites. It is common to have a combination of various types of networks.

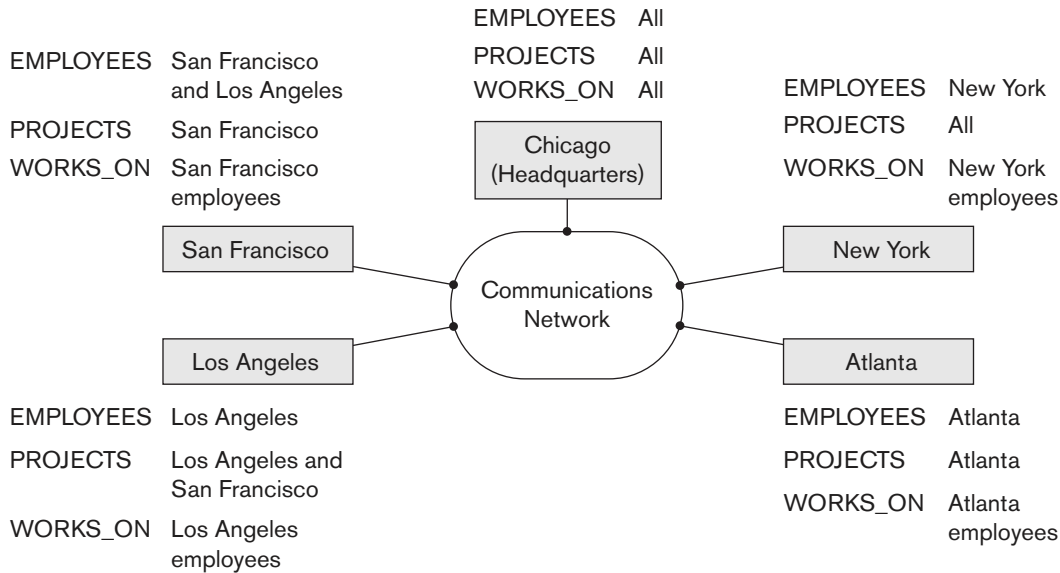
Networks may have different **topologies** that define the direct communication paths among sites. The type and topology of the network used may have a significant impact on the performance and hence on the strategies for distributed query processing and distributed database design. For high-level architectural issues, however, it does not matter what type of network is used; what matters is that each site be able to communicate, directly or indirectly, with every other site. For the remainder of this chapter, we assume that some type of network exists among nodes, regardless of any particular topology. We will not address any network-specific issues, although it is important to understand that for an efficient operation of a distributed database system (DDBS), network design and performance issues are critical and are an integral part of the overall solution. The details of the underlying network are invisible to the end user.

### 23.1.2 Transparency

The concept of transparency extends the general idea of hiding implementation details from end users. A highly transparent system offers a lot of flexibility to the end user/application developer since it requires little or no awareness of underlying details on their part. In the case of a traditional centralized database, transparency simply pertains to logical and physical data independence for application developers. However, in a DDB scenario, the data and software are distributed over multiple nodes connected by a computer network, so additional types of transparencies are introduced.

Consider the company database in Figure 5.5 that we have been discussing throughout the book. The `EMPLOYEE`, `PROJECT`, and `WORKS_ON` tables may be fragmented horizontally (that is, into sets of rows, as we will discuss in Section 23.2) and stored with possible replication, as shown in Figure 23.1. The following types of transparencies are possible:

- **Data organization transparency (also known as *distribution or network transparency*).** This refers to freedom for the user from the operational details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency. **Location transparency** refers to the fact that the command used to perform a task is independent of the location of the data and the location of the node where the command was issued. **Naming transparency** implies that once a name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.
- **Replication transparency.** As we show in Figure 23.1, copies of the same data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.
- **Fragmentation transparency.** Two types of fragmentation are possible. **Horizontal fragmentation** distributes a relation (table) into subrelations that are subsets of the tuples (rows) in the original relation; this is also known



**Figure 23.1**  
Data distribution and replication among distributed databases.

as **sharding** in the newer big data and cloud computing systems. **Vertical fragmentation** distributes a relation into subrelations where each subrelation is defined by a subset of the columns of the original relation. Fragmentation transparency makes the user unaware of the existence of fragments.

- Other transparencies include **design transparency** and **execution transparency**—which refer, respectively, to freedom from knowing how the distributed database is designed and where a transaction executes.

### 23.1.3 Availability and Reliability

Reliability and availability are two of the most common potential advantages cited for distributed databases. **Reliability** is broadly defined as the probability that a system is running (not down) at a certain time point, whereas **availability** is the probability that the system is continuously available during a time interval. We can directly relate reliability and availability of the database to the faults, errors, and failures associated with it. A failure can be described as a deviation of a system's behavior from that which is specified in order to ensure correct execution of operations. **Errors** constitute that subset of system states that causes the failure. **Fault** is the cause of an error.

To construct a system that is reliable, we can adopt several approaches. One common approach stresses *fault tolerance*; it recognizes that faults will occur, and it designs mechanisms that can detect and remove faults before they can result in a

system failure. Another more stringent approach attempts to ensure that the final system does not contain any faults. This is done through an exhaustive design process followed by extensive quality control and testing. A reliable DDBMS tolerates failures of underlying components, and it processes user requests as long as database consistency is not violated. A DDBMS recovery manager has to deal with failures arising from transactions, hardware, and communication networks. Hardware failures can either be those that result in loss of main memory contents or loss of secondary storage contents. Network failures occur due to errors associated with messages and line failures. Message errors can include their loss, corruption, or out-of-order arrival at destination.

The previous definitions are used in computer systems in general, where there is a technical distinction between reliability and availability. In most discussions related to DDB, the term **availability** is used generally as an umbrella term to cover both concepts.

### 23.1.4 Scalability and Partition Tolerance

**Scalability** determines the extent to which the system can expand its capacity while continuing to operate without interruption. There are two types of scalability:

1. **Horizontal scalability:** This refers to expanding the number of nodes in the distributed system. As nodes are added to the system, it should be possible to distribute some of the data and processing loads from existing nodes to the new nodes.
2. **Vertical scalability:** This refers to expanding the capacity of the individual nodes in the system, such as expanding the storage capacity or the processing power of a node.

As the system expands its number of nodes, it is possible that the network, which connects the nodes, may have faults that cause the nodes to be partitioned into groups of nodes. The nodes within each partition are still connected by a subnetwork, but communication among the partitions is lost. The concept of **partition tolerance** states that the system should have the capacity to continue operating while the network is partitioned.

### 23.1.5 Autonomy

**Autonomy** determines the extent to which individual nodes or DBs in a connected DDB can operate independently. A high degree of autonomy is desirable for increased flexibility and customized maintenance of an individual node. Autonomy can be applied to design, communication, and execution. **Design autonomy** refers to independence of data model usage and transaction management techniques among nodes. **Communication autonomy** determines the extent to which each node can decide on sharing of information with other nodes. **Execution autonomy** refers to independence of users to act as they please.



### 23.1.6 Advantages of Distributed Databases

Some important advantages of DDB are listed below.

1. **Improved ease and flexibility of application development.** Developing and maintaining applications at geographically distributed sites of an organization is facilitated due to transparency of data distribution and control.
2. **Increased availability.** This is achieved by the isolation of faults to their site of origin without affecting the other database nodes connected to the network. When the data and DDBMS software are distributed over many sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. Further improvement is achieved by judiciously replicating data and software at more than one site. In a centralized system, failure at a single site makes the whole system unavailable to all users. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database. If the data in the failed site has been replicated at another site prior to the failure, then the user will not be affected at all. The ability of the system to survive network partitioning also contributes to high availability.
3. **Improved performance.** A distributed DBMS fragments the database by keeping the data closer to where it is needed most. **Data localization** reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks. When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local databases. In addition, each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. Moreover, interquery and intraquery parallelism can be achieved by executing multiple queries at different sites, or by breaking up a query into a number of subqueries that execute in parallel. This contributes to improved performance.
4. **Easier expansion via scalability.** In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more nodes is much easier than in centralized (non-distributed) systems.

The transparencies we discussed in Section 23.1.2 lead to a compromise between ease of use and the overhead cost of providing transparency. Total transparency provides the global user with a view of the entire DDBS as if it is a single centralized system. Transparency is provided as a complement to **autonomy**, which gives the users tighter control over local databases. Transparency features may be implemented as a part of the user language, which may translate the required services into appropriate operations.

## 23.2 Data Fragmentation, Replication, and Allocation Techniques for Distributed Database Design

In this section, we discuss techniques that are used to break up the database into logical units, called **fragments**, which may be assigned for storage at the various nodes. We also discuss the use of **data replication**, which permits certain data to be stored in more than one site to increase availability and reliability; and the process of **allocating** fragments—or replicas of fragments—for storage at the various nodes. These techniques are used during the process of **distributed database design**. The information concerning data fragmentation, allocation, and replication is stored in a **global directory** that is accessed by the DDBS applications as needed.

### 23.2.1 Data Fragmentation and Sharding

In a DDB, decisions must be made regarding which site should be used to store which portions of the database. For now, we will assume that there is *no replication*; that is, each relation—or portion of a relation—is stored at one site only. We discuss replication and its effects later in this section. We also use the terminology of relational databases, but similar concepts apply to other data models. We assume that we are starting with a relational database schema and must decide on how to distribute the relations over the various sites. To illustrate our discussion, we use the relational database schema shown in Figure 5.5.

Before we decide on how to distribute the data, we must determine the *logical units* of the database that are to be distributed. The simplest logical units are the relations themselves; that is, each *whole* relation is to be stored at a particular site. In our example, we must decide on a site to store each of the relations EMPLOYEE, DEPARTMENT, PROJECT, WORKS\_ON, and DEPENDENT in Figure 5.5. In many cases, however, a relation can be divided into smaller logical units for distribution. For example, consider the company database shown in Figure 5.6, and assume there are three computer sites—one for each department in the company.<sup>1</sup>

We may want to store the database information relating to each department at the computer site for that department. A technique called *horizontal fragmentation* or *sharding* can be used to partition each relation by department.

**Horizontal Fragmentation (Sharding).** A **horizontal fragment** or **shard** of a relation is a subset of the tuples in that relation. The tuples that belong to the horizontal fragment can be specified by a condition on one or more attributes of the relation, or by some other mechanism. Often, only a single attribute is involved in the condition. For example, we may define three horizontal fragments on the EMPLOYEE relation in Figure 5.6 with the following conditions: (Dno = 5), (Dno = 4), and (Dno = 1)—each

---

<sup>1</sup>Of course, in an actual situation, there will be many more tuples in the relation than those shown in Figure 5.6.

fragment contains the EMPLOYEE tuples working for a particular department. Similarly, we may define three horizontal fragments for the PROJECT relation, with the conditions ( $Dnum = 5$ ), ( $Dnum = 4$ ), and ( $Dnum = 1$ )—each fragment contains the PROJECT tuples controlled by a particular department. **Horizontal fragmentation** divides a relation *horizontally* by grouping rows to create subsets of tuples, where each subset has a certain logical meaning. These fragments can then be assigned to different sites (nodes) in the distributed system. **Derived horizontal fragmentation** applies the partitioning of a primary relation (DEPARTMENT in our example) to other secondary relations (EMPLOYEE and PROJECT in our example), which are related to the primary via a foreign key. Thus, related data between the primary and the secondary relations gets fragmented in the same way.

**Vertical Fragmentation.** Each site may not need all the attributes of a relation, which would indicate the need for a different type of fragmentation. **Vertical fragmentation** divides a relation “vertically” by columns. A **vertical fragment** of a relation keeps only certain attributes of the relation. For example, we may want to fragment the EMPLOYEE relation into two vertical fragments. The first fragment includes personal information—Name, Bdate, Address, and Sex—and the second includes work-related information—Ssn, Salary, Super\_ssn, and Dno. This vertical fragmentation is not quite proper, because if the two fragments are stored separately, we cannot put the original employee tuples back together since there is *no common attribute* between the two fragments. It is necessary to include the primary key or some unique key attribute in *every* vertical fragment so that the full relation can be reconstructed from the fragments. Hence, we must add the Ssn attribute to the personal information fragment.

Notice that each horizontal fragment on a relation  $R$  can be specified in the relational algebra by a  $\sigma_{C_i}(R)$  (select) operation. A set of horizontal fragments whose conditions  $C_1, C_2, \dots, C_n$  include all the tuples in  $R$ —that is, every tuple in  $R$  satisfies  $(C_1 \text{ OR } C_2 \text{ OR } \dots \text{ OR } C_n)$ —is called a **complete horizontal fragmentation** of  $R$ . In many cases a complete horizontal fragmentation is also **disjoint**; that is, no tuple in  $R$  satisfies  $(C_i \text{ AND } C_j)$  for any  $i \neq j$ . Our two earlier examples of horizontal fragmentation for the EMPLOYEE and PROJECT relations were both complete and disjoint. To reconstruct the relation  $R$  from a *complete* horizontal fragmentation, we need to apply the UNION operation to the fragments.

A vertical fragment on a relation  $R$  can be specified by a  $\pi_{L_i}(R)$  operation in the relational algebra. A set of vertical fragments whose projection lists  $L_1, L_2, \dots, L_n$  include all the attributes in  $R$  but share only the primary key attribute of  $R$  is called a **complete vertical fragmentation** of  $R$ . In this case the projection lists satisfy the following two conditions:

- $L_1 \cup L_2 \cup \dots \cup L_n = \text{ATTRS}(R)$
- $L_i \cap L_j = \text{PK}(R)$  for any  $i \neq j$ , where  $\text{ATTRS}(R)$  is the set of attributes of  $R$  and  $\text{PK}(R)$  is the primary key of  $R$

To reconstruct the relation  $R$  from a *complete* vertical fragmentation, we apply the OUTER UNION operation to the vertical fragments (assuming no horizontal

fragmentation is used). Notice that we could also apply a FULL OUTER JOIN operation and get the same result for a complete vertical fragmentation, even when some horizontal fragmentation may also have been applied. The two vertical fragments of the EMPLOYEE relation with projection lists  $L_1 = \{\text{Ssn, Name, Bdate, Address, Sex}\}$  and  $L_2 = \{\text{Ssn, Salary, Super\_ssn, Dno}\}$  constitute a complete vertical fragmentation of EMPLOYEE.

Two horizontal fragments that are neither complete nor disjoint are those defined on the EMPLOYEE relation in Figure 5.5 by the conditions ( $\text{Salary} > 50000$ ) and ( $\text{Dno} = 4$ ); they may not include all EMPLOYEE tuples, and they may include common tuples. Two vertical fragments that are not complete are those defined by the attribute lists  $L_1 = \{\text{Name, Address}\}$  and  $L_2 = \{\text{Ssn, Name, Salary}\}$ ; these lists violate both conditions of a complete vertical fragmentation.

**Mixed (Hybrid) Fragmentation.** We can intermix the two types of fragmentation, yielding a **mixed fragmentation**. For example, we may combine the horizontal and vertical fragmentations of the EMPLOYEE relation given earlier into a mixed fragmentation that includes six fragments. In this case, the original relation can be reconstructed by applying UNION *and* OUTER UNION (or OUTER JOIN) operations in the appropriate order. In general, a **fragment** of a relation  $R$  can be specified by a SELECT-PROJECT combination of operations  $\pi_L(\sigma_C(R))$ . If  $C = \text{TRUE}$  (that is, all tuples are selected) and  $L \neq \text{ATTRS}(R)$ , we get a vertical fragment, and if  $C \neq \text{TRUE}$  and  $L = \text{ATTRS}(R)$ , we get a horizontal fragment. Finally, if  $C \neq \text{TRUE}$  and  $L \neq \text{ATTRS}(R)$ , we get a mixed fragment. Notice that a relation can itself be considered a fragment with  $C = \text{TRUE}$  and  $L = \text{ATTRS}(R)$ . In the following discussion, the term *fragment* is used to refer to a relation or to any of the preceding types of fragments.

A **fragmentation schema** of a database is a definition of a set of fragments that includes *all* attributes and tuples in the database and satisfies the condition that the whole database can be reconstructed from the fragments by applying some sequence of OUTER UNION (or OUTER JOIN) and UNION operations. It is also sometimes useful—although not necessary—to have all the fragments be disjoint except for the repetition of primary keys among vertical (or mixed) fragments. In the latter case, all replication and distribution of fragments is clearly specified at a subsequent stage, separately from fragmentation.

An **allocation schema** describes the allocation of fragments to nodes (sites) of the DDBS; hence, it is a mapping that specifies for each fragment the site(s) at which it is stored. If a fragment is stored at more than one site, it is said to be **replicated**. We discuss data replication and allocation next.

### 23.2.2 Data Replication and Allocation

Replication is useful in improving the availability of data. The most extreme case is replication of the *whole database* at every site in the distributed system, thus creating a **fully replicated distributed database**. This can improve availability remarkably because the system can continue to operate as long as at least one site is up. It

also improves performance of retrieval (read performance) for global queries because the results of such queries can be obtained locally from any one site; hence, a retrieval query can be processed at the local site where it is submitted, if that site includes a server module. The disadvantage of full replication is that it can slow down update operations (write performance) drastically, since a single logical update must be performed on *every copy* of the database to keep the copies consistent. This is especially true if many copies of the database exist. Full replication makes the concurrency control and recovery techniques more expensive than they would be if there was no replication, as we will see in Section 23.3.

The other extreme from full replication involves having **no replication**—that is, each fragment is stored at exactly one site. In this case, all fragments *must be* disjoint, except for the repetition of primary keys among vertical (or mixed) fragments. This is also called **nonredundant allocation**.

Between these two extremes, we have a wide spectrum of **partial replication** of the data—that is, some fragments of the database may be replicated whereas others may not. The number of copies of each fragment can range from one up to the total number of sites in the distributed system. A special case of partial replication is occurring heavily in applications where mobile workers—such as sales forces, financial planners, and claims adjusters—carry partially replicated databases with them on laptops and PDAs and synchronize them periodically with the server database. A description of the replication of fragments is sometimes called a **replication schema**.

Each fragment—or each copy of a fragment—must be assigned to a particular site in the distributed system. This process is called **data distribution** (or **data allocation**). The choice of sites and the degree of replication depend on the performance and availability goals of the system and on the types and frequencies of transactions submitted at each site. For example, if high availability is required, transactions can be submitted at any site, and most transactions are retrieval only, a fully replicated database is a good choice. However, if certain transactions that access particular parts of the database are mostly submitted at a particular site, the corresponding set of fragments can be allocated at that site only. Data that is accessed at multiple sites can be replicated at those sites. If many updates are performed, it may be useful to limit replication. Finding an optimal or even a good solution to distributed data allocation is a complex optimization problem.

### 23.2.3 Example of Fragmentation, Allocation, and Replication

We now consider an example of fragmenting and distributing the company database in Figures 5.5 and 5.6. Suppose that the company has three computer sites—one for each current department. Sites 2 and 3 are for departments 5 and 4, respectively. At each of these sites, we expect frequent access to the EMPLOYEE and PROJECT information for the employees *who work in that department* and the projects *controlled by that department*. Further, we assume that these sites mainly access the Name, Ssn, Salary, and Super\_ssn attributes of EMPLOYEE. Site 1 is used

by company headquarters and accesses all employee and project information regularly, in addition to keeping track of DEPENDENT information for insurance purposes.

According to these requirements, the whole database in Figure 5.6 can be stored at site 1. To determine the fragments to be replicated at sites 2 and 3, first we can horizontally fragment DEPARTMENT by its key Dnumber. Then we apply derived fragmentation to the EMPLOYEE, PROJECT, and DEPT\_LOCATIONS relations based on their foreign keys for department number—called Dno, Dnum, and Dnumber, respectively, in Figure 5.5. We can vertically fragment the resulting EMPLOYEE fragments to include only the attributes {Name, Ssn, Salary, Super\_ssn, Dno}. Figure 23.2 shows the mixed fragments EMPD\_5 and EMPD\_4, which include the EMPLOYEE tuples satisfying the conditions  $Dno = 5$  and  $Dno = 4$ , respectively. The horizontal fragments of PROJECT, DEPARTMENT, and DEPT\_LOCATIONS are similarly fragmented by department number. All these fragments—stored at sites 2 and 3—are replicated because they are also stored at headquarters—site 1.

We must now fragment the WORKS\_ON relation and decide which fragments of WORKS\_ON to store at sites 2 and 3. We are confronted with the problem that no attribute of WORKS\_ON directly indicates the department to which each tuple belongs. In fact, each tuple in WORKS\_ON relates an employee  $e$  to a project  $P$ . We could fragment WORKS\_ON based on the department  $D$  in which  $e$  works or based on the department  $D'$  that controls  $P$ . Fragmentation becomes easy if we have a constraint stating that  $D = D'$  for all WORKS\_ON tuples—that is, if employees can work only on projects controlled by the department they work for. However, there is no such constraint in our database in Figure 5.6. For example, the WORKS\_ON tuple  $\langle 333445555, 10, 10.0 \rangle$  relates an employee who works for department 5 with a project controlled by department 4. In this case, we could fragment WORKS\_ON based on the department in which the employee works (which is expressed by the condition C) and then fragment further based on the department that controls the projects that employee is working on, as shown in Figure 23.3.

In Figure 23.3, the union of fragments  $G_1$ ,  $G_2$ , and  $G_3$  gives all WORKS\_ON tuples for employees who work for department 5. Similarly, the union of fragments  $G_4$ ,  $G_5$ , and  $G_6$  gives all WORKS\_ON tuples for employees who work for department 4. On the other hand, the union of fragments  $G_1$ ,  $G_4$ , and  $G_7$  gives all WORKS\_ON tuples for projects controlled by department 5. The condition for each of the fragments  $G_1$  through  $G_9$  is shown in Figure 23.3. The relations that represent M:N relationships, such as WORKS\_ON, often have several possible logical fragmentations. In our distribution in Figure 23.2, we choose to include all fragments that can be joined to either an EMPLOYEE tuple or a PROJECT tuple at sites 2 and 3. Hence, we place the union of fragments  $G_1$ ,  $G_2$ ,  $G_3$ ,  $G_4$ , and  $G_7$  at site 2 and the union of fragments  $G_4$ ,  $G_5$ ,  $G_6$ ,  $G_2$ , and  $G_8$  at site 3. Notice that fragments  $G_2$  and  $G_4$  are replicated at both sites. This allocation strategy permits the join between the local EMPLOYEE or PROJECT fragments at site 2 or site 3 and the local WORKS\_ON fragment to be performed completely locally. This clearly demonstrates how complex the problem of database fragmentation and allocation is for large databases. The Selected Bibliography at the end of this chapter discusses some of the work done in this area.

**(a) EMPD\_5**

Fname	Minit	Lname	<u>Ssn</u>	Salary	Super_ssn	Dno
John	B	Smith	123456789	30000	333445555	5
Franklin	T	Wong	333445555	40000	888665555	5
Ramesh	K	Narayan	666884444	38000	333445555	5
Joyce	A	English	453453453	25000	333445555	5

**DEP\_5**

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22

**DEP\_5\_LOCS**

<u>Dnumber</u>	<u>Location</u>
5	Bellaire
5	Sugarland
5	Houston

**WORKS\_ON\_5**

<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0

**PROJS\_5**

Pname	<u>Pnumber</u>	Plocation	Dnum
Product X	1	Bellaire	5
Product Y	2	Sugarland	5
Product Z	3	Houston	5

**Data at site 2****(b) EMPD\_4**

Fname	Minit	Lname	<u>Ssn</u>	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	25000	987654321	4
Jennifer	S	Wallace	987654321	43000	888665555	4
Ahmad	V	Jabbar	987987987	25000	987654321	4

**DEP\_4**

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
Administration	4	987654321	1995-01-01

**DEP\_4\_LOCS**

<u>Dnumber</u>	<u>Location</u>
4	Stafford

**WORKS\_ON\_4**

<u>Essn</u>	<u>Pno</u>	Hours
333445555	10	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0

**PROJS\_4**

Pname	<u>Pnumber</u>	Plocation	Dnum
Computerization	10	Stafford	4
New_benefits	30	Stafford	4

**Data at site 3****Figure 23.2**

Allocation of fragments to sites. (a) Relation fragments at site 2 corresponding to department 5. (b) Relation fragments at site 3 corresponding to department 4.



**Figure 23.3**

Complete and disjoint fragments of the WORKS\_ON relation. (a) Fragments of WORKS\_ON for employees working in department 5 ( $C = [Essn \text{ in } (SELECT Ssn \text{ FROM EMPLOYEE WHERE Dno} = 5)]$ ). (b) Fragments of WORKS\_ON for employees working in department 4 ( $C = [Essn \text{ in } (SELECT Ssn \text{ FROM EMPLOYEE WHERE Dno} = 4)]$ ). (c) Fragments of WORKS\_ON for employees working in department 1 ( $C = [Essn \text{ in } (SELECT Ssn \text{ FROM EMPLOYEE WHERE Dno} = 1)]$ ).

**(a) Employees in Department 5****G1**

<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0

$C1 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 5))$

**G2**

<u>Essn</u>	<u>Pno</u>	Hours
333445555	10	10.0

$C2 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 4))$

**G3**

<u>Essn</u>	<u>Pno</u>	Hours
333445555	20	10.0

$C3 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 1))$

**(b) Employees in Department 4****G4**

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

$C4 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 5))$

**G5**

<u>Essn</u>	<u>Pno</u>	Hours
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0

$C5 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 4))$

**G6**

<u>Essn</u>	<u>Pno</u>	Hours
987654321	20	15.0

$C6 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 1))$

**(c) Employees in Department 1****G7**

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

$C7 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 5))$

**G8**

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

$C8 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 4))$

**G9**

<u>Essn</u>	<u>Pno</u>	Hours
888665555	20	Null

$C9 = C \text{ and } (Pno \text{ in } (SELECT Pnumber \text{ FROM PROJECT WHERE Dnum} = 1))$



## 23.3 Overview of Concurrency Control and Recovery in Distributed Databases

For concurrency control and recovery purposes, numerous problems arise in a distributed DBMS environment that are not encountered in a centralized DBMS environment. These include the following:

- **Dealing with *multiple copies of the data items*.** The concurrency control method is responsible for maintaining consistency among these copies. The recovery method is responsible for making a copy consistent with other copies if the site on which the copy is stored fails and recovers later.
- **Failure of individual sites.** The DDBMS should continue to operate with its running sites, if possible, when one or more individual sites fail. When a site recovers, its local database must be brought up-to-date with the rest of the sites before it rejoins the system.
- **Failure of communication links.** The system must be able to deal with the failure of one or more of the communication links that connect the sites. An extreme case of this problem is that **network partitioning** may occur. This breaks up the sites into two or more partitions, where the sites within each partition can communicate only with one another and not with sites in other partitions.
- **Distributed commit.** Problems can arise with committing a transaction that is accessing databases stored on multiple sites if some sites fail during the commit process. The **two-phase commit protocol** (see Section 21.6) is often used to deal with this problem.
- **Distributed deadlock.** Deadlock may occur among several sites, so techniques for dealing with deadlocks must be extended to take this into account.

Distributed concurrency control and recovery techniques must deal with these and other problems. In the following subsections, we review some of the techniques that have been suggested to deal with recovery and concurrency control in DDBMSs.

### 23.3.1 Distributed Concurrency Control Based on a Distinguished Copy of a Data Item

To deal with replicated data items in a distributed database, a number of concurrency control methods have been proposed that extend the concurrency control techniques that are used in centralized databases. We discuss these techniques in the context of extending centralized *locking*. Similar extensions apply to other concurrency control techniques. The idea is to designate *a particular copy* of each data item as a **distinguished copy**. The locks for this data item are associated *with the distinguished copy*, and all locking and unlocking requests are sent to the site that contains that copy.

A number of different methods are based on this idea, but they differ in their method of choosing the distinguished copies. In the **primary site technique**, all distinguished copies are kept at the same site. A modification of this approach is the primary site with a **backup site**. Another approach is the **primary copy** method, where the distinguished copies of the various data items can be stored in different sites. A site that includes a distinguished copy of a data item basically acts as the **coordinator site** for concurrency control on that item. We discuss these techniques next.

**Primary Site Technique.** In this method, a single **primary site** is designated to be the **coordinator site** for *all database items*. Hence, all locks are kept at that site, and all requests for locking or unlocking are sent there. This method is thus an extension of the centralized locking approach. For example, if all transactions follow the two-phase locking protocol, serializability is guaranteed. The advantage of this approach is that it is a simple extension of the centralized approach and thus is not overly complex. However, it has certain inherent disadvantages. One is that all locking requests are sent to a single site, possibly overloading that site and causing a system bottleneck. A second disadvantage is that failure of the primary site paralyzes the system, since all locking information is kept at that site. This can limit system reliability and availability.

Although all locks are accessed at the primary site, the items themselves can be accessed at any site at which they reside. For example, once a transaction obtains a `Read_lock` on a data item from the primary site, it can access any copy of that data item. However, once a transaction obtains a `Write_lock` and updates a data item, the DDBMS is responsible for updating *all copies* of the data item before releasing the lock.

**Primary Site with Backup Site.** This approach addresses the second disadvantage of the primary site method by designating a second site to be a **backup site**. All locking information is maintained at both the primary and the backup sites. In case of primary site failure, the backup site takes over as the primary site, and a new backup site is chosen. This simplifies the process of recovery from failure of the primary site, since the backup site takes over and processing can resume after a new backup site is chosen and the lock status information is copied to that site. It slows down the process of acquiring locks, however, because all lock requests and granting of locks must be recorded at *both the primary and the backup sites* before a response is sent to the requesting transaction. The problem of the primary and backup sites becoming overloaded with requests and slowing down the system remains undiminished.

**Primary Copy Technique.** This method attempts to distribute the load of lock coordination among various sites by having the distinguished copies of different data items *stored at different sites*. Failure of one site affects any transactions that are accessing locks on items whose primary copies reside at that site, but other transactions are not affected. This method can also use backup sites to enhance reliability and availability.

**Choosing a New Coordinator Site in Case of Failure.** Whenever a coordinator site fails in any of the preceding techniques, the sites that are still running must choose a new coordinator. In the case of the primary site approach with *no* backup site, all executing transactions must be aborted and restarted in a tedious recovery process. Part of the recovery process involves choosing a new primary site and creating a lock manager process and a record of all lock information at that site. For methods that use backup sites, transaction processing is suspended while the backup site is designated as the new primary site and a new backup site is chosen and is sent copies of all the locking information from the new primary site.

If a backup site *X* is about to become the new primary site, *X* can choose the new backup site from among the system's running sites. However, if no backup site existed, or if both the primary and the backup sites are down, a process called **election** can be used to choose the new coordinator site. In this process, any site *Y* that attempts to communicate with the coordinator site repeatedly and fails to do so can assume that the coordinator is down and can start the election process by sending a message to all running sites proposing that *Y* become the new coordinator. As soon as *Y* receives a majority of yes votes, *Y* can declare that it is the new coordinator. The election algorithm itself is complex, but this is the main idea behind the election method. The algorithm also resolves any attempt by two or more sites to become coordinator at the same time. The references in the Selected Bibliography at the end of this chapter discuss the process in detail.

### 23.3.2 Distributed Concurrency Control Based on Voting

The concurrency control methods for replicated items discussed earlier all use the idea of a distinguished copy that maintains the locks for that item. In the **voting method**, there is no distinguished copy; rather, a lock request is sent to all sites that includes a copy of the data item. Each copy maintains its own lock and can grant or deny the request for it. If a transaction that requests a lock is granted that lock by *a majority* of the copies, it holds the lock and informs *all copies* that it has been granted the lock. If a transaction does not receive a majority of votes granting it a lock within a certain *time-out period*, it cancels its request and informs all sites of the cancellation.

The voting method is considered a truly distributed concurrency control method, since the responsibility for a decision resides with all the sites involved. Simulation studies have shown that voting has higher message traffic among sites than do the distinguished copy methods. If the algorithm takes into account possible site failures during the voting process, it becomes extremely complex.

### 23.3.3 Distributed Recovery

The recovery process in distributed databases is quite involved. We give only a very brief idea of some of the issues here. In some cases it is difficult even to determine whether a site is down without exchanging numerous messages with other sites. For

example, suppose that site *X* sends a message to site *Y* and expects a response from *Y* but does not receive it. There are several possible explanations:

- The message was not delivered to *Y* because of communication failure.
- Site *Y* is down and could not respond.
- Site *Y* is running and sent a response, but the response was not delivered.

Without additional information or the sending of additional messages, it is difficult to determine what actually happened.

Another problem with distributed recovery is distributed commit. When a transaction is updating data at several sites, it cannot commit until it is sure that the effect of the transaction on *every* site cannot be lost. This means that every site must first have recorded the local effects of the transactions permanently in the local site log on disk. The two-phase commit protocol is often used to ensure the correctness of distributed commit (see Section 21.6).

## 23.4 Overview of Transaction Management in Distributed Databases

The global and local transaction management software modules, along with the concurrency control and recovery manager of a DDBMS, collectively guarantee the ACID properties of transactions (see Chapter 20).

An additional component called the **global transaction manager** is introduced for supporting distributed transactions. The site where the transaction originated can temporarily assume the role of global transaction manager and coordinate the execution of database operations with transaction managers across multiple sites. Transaction managers export their functionality as an interface to the application programs. The operations exported by this interface are similar to those covered in Section 20.2.1, namely `BEGIN_TRANSACTION`, `READ` or `WRITE`, `END_TRANSACTION`, `COMMIT_TRANSACTION`, and `ROLLBACK` (or `ABORT`). The manager stores book-keeping information related to each transaction, such as a unique identifier, originating site, name, and so on. For `READ` operations, it returns a local copy if valid and available. For `WRITE` operations, it ensures that updates are visible across all sites containing copies (replicas) of the data item. For `ABORT` operations, the manager ensures that no effects of the transaction are reflected in any site of the distributed database. For `COMMIT` operations, it ensures that the effects of a write are persistently recorded on all databases containing copies of the data item. Atomic termination (`COMMIT`/ `ABORT`) of distributed transactions is commonly implemented using the two-phase commit protocol (see Section 22.6).

The transaction manager passes to the concurrency controller module the database operations and associated information. The controller is responsible for acquisition and release of associated locks. If the transaction requires access to a locked resource, it is blocked until the lock is acquired. Once the lock is acquired, the operation is sent to the runtime processor, which handles the actual execution of the

database operation. Once the operation is completed, locks are released and the transaction manager is updated with the result of the operation.

### 23.4.1 Two-Phase Commit Protocol

In Section 22.6, we described the *two-phase commit protocol (2PC)*, which requires a **global recovery manager**, or **coordinator**, to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables). The two-phase commit protocol has certain drawbacks that led to the development of the three-phase commit protocol, which we discuss next.

### 23.4.2 Three-Phase Commit Protocol

The biggest drawback of 2PC is that it is a blocking protocol. Failure of the coordinator blocks all participating sites, causing them to wait until the coordinator recovers. This can cause performance degradation, especially if participants are holding locks to shared resources. Other types of problems may also occur that make the outcome of the transaction nondeterministic.

These problems are solved by the three-phase commit (3PC) protocol, which essentially divides the second commit phase into two subphases called **prepare-to-commit** and **commit**. The prepare-to-commit phase is used to communicate the result of the vote phase to all participants. If all participants vote yes, then the coordinator instructs them to move into the prepare-to-commit state. The commit subphase is identical to its two-phase counterpart. Now, if the coordinator crashes during this subphase, another participant can see the transaction through to completion. It can simply ask a crashed participant if it received a prepare-to-commit message. If it did not, then it safely assumes to abort. Thus the state of the protocol can be recovered irrespective of which participant crashes. Also, by limiting the time required for a transaction to commit or abort to a maximum time-out period, the protocol ensures that a transaction attempting to commit via 3PC releases locks on time-out.

The main idea is to limit the wait time for participants who have prepared to commit and are waiting for a global commit or abort from the coordinator. When a participant receives a precommit message, it knows that the rest of the participants have voted to commit. If a precommit message has not been received, then the participant will abort and release all locks.

### 23.4.3 Operating System Support for Transaction Management

The following are the main benefits of operating system (OS)-supported transaction management:

- Typically, DBMSs use their own semaphores<sup>2</sup> to guarantee mutually exclusive access to shared resources. Since these semaphores are implemented in

---

<sup>2</sup>Semaphores are data structures used for synchronized and exclusive access to shared resources for preventing race conditions in a parallel computing system.

user space at the level of the DBMS application software, the OS has no knowledge about them. Hence if the OS deactivates a DBMS process holding a lock, other DBMS processes wanting this locked resource get blocked. Such a situation can cause serious performance degradation. OS-level knowledge of semaphores can help eliminate such situations.

- Specialized hardware support for locking can be exploited to reduce associated costs. This can be of great importance, since locking is one of the most common DBMS operations.
- Providing a set of common transaction support operations though the kernel allows application developers to focus on adding new features to their products as opposed to reimplementing the common functionality for each application. For example, if different DDBMSs are to coexist on the same machine and they chose the two-phase commit protocol, then it is more beneficial to have this protocol implemented as part of the kernel so that the DDBMS developers can focus more on adding new features to their products.

## 23.5 Query Processing and Optimization in Distributed Databases

Now we give an overview of how a DDBMS processes and optimizes a query. First we discuss the steps involved in query processing and then elaborate on the communication costs of processing a distributed query. Then we discuss a special operation, called a *semijoin*, which is used to optimize some types of queries in a DDBMS. A detailed discussion about optimization algorithms is beyond the scope of this text. We attempt to illustrate optimization principles using suitable examples.<sup>3</sup>

### 23.5.1 Distributed Query Processing

A distributed database query is processed in stages as follows:

1. **Query Mapping.** The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations. This translation is done by referring to the global conceptual schema and does not take into account the actual distribution and replication of data. Hence, this translation is largely identical to the one performed in a centralized DBMS. It is first normalized, analyzed for semantic errors, simplified, and finally restructured into an algebraic query.
2. **Localization.** In a distributed database, fragmentation results in relations being stored in separate sites, with some fragments possibly being replicated. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.

---

<sup>3</sup>For a detailed discussion of optimization algorithms, see Ozsu and Valduriez (1999).

3. **Global Query Optimization.** Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs. Since DDBs are connected by a network, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).
4. **Local Query Optimization.** This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

The first three stages discussed above are performed at a central control site, whereas the last stage is performed locally.

### 23.5.2 Data Transfer Costs of Distributed Query Processing

We discussed the issues involved in processing and optimizing a query in a centralized DBMS in Chapter 19. In a distributed system, several additional factors further complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred to the site where the query result is needed. Although these costs may not be very high if the sites are connected via a high-performance local area network, they become significant in other types of networks. Hence, DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy.

We illustrate this with two simple sample queries. Suppose that the EMPLOYEE and DEPARTMENT relations in Figure 3.5 are distributed at two sites as shown in Figure 23.4. We will assume in this example that neither relation is fragmented. According to Figure 23.4, the size of the EMPLOYEE relation is  $100 \times 10,000 = 10^6$  bytes, and the size of the DEPARTMENT relation is  $35 \times 100 = 3,500$  bytes. Consider the query Q: *For each employee, retrieve the employee name and the name of the department for which the employee works.* This can be stated as follows in the relational algebra:

$$Q: \pi_{Fname, Lname, Dname} (EMPLOYEE \bowtie_{Dno=Dnumber} DEPARTMENT)$$

The result of this query will include 10,000 records, assuming that every employee is related to a department. Suppose that each record in the query result is *40 bytes long*. The query is submitted at a distinct site 3, which is called the **result site** because the query result is needed there. Neither the EMPLOYEE nor the DEPARTMENT relations reside at site 3. There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of  $1,000,000 + 3,500 = 1,003,500$  bytes must be transferred.



Site 1:

**EMPLOYEE**

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

10,000 records

each record is 100 bytes long

Ssn field is 9 bytes long

Dno field is 4 bytes long

Fname field is 15 bytes long

Lname field is 15 bytes long

Site 2:

**DEPARTMENT**

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

100 records

each record is 35 bytes long

Dnumber field is 4 bytes long

Mgr\_ssn field is 9 bytes long

Dname field is 10 bytes long

**Figure 23.4**

Example to illustrate volume of data transferred.

2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is  $40 \times 10,000 = 400,000$  bytes, so  $400,000 + 1,000,000 = 1,400,000$  bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case,  $400,000 + 3,500 = 403,500$  bytes must be transferred.

If minimizing the amount of data transfer is our optimization criterion, we should choose strategy 3. Now consider another query  $Q'$ : *For each department, retrieve the department name and the name of the department manager.* This can be stated as follows in the relational algebra:

$$Q': \pi_{Fname, Lname, Dname} (DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE)$$

Again, suppose that the query is submitted at site 3. The same three strategies for executing query  $Q$  apply to  $Q'$ , except that the result of  $Q'$  includes only 100 records, assuming that each department has a manager:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of  $1,000,000 + 3,500 = 1,003,500$  bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is  $40 \times 100 = 4,000$  bytes, so  $4,000 + 1,000,000 = 1,004,000$  bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case,  $4,000 + 3,500 = 7,500$  bytes must be transferred.

Again, we would choose strategy 3—this time by an overwhelming margin over strategies 1 and 2. The preceding three strategies are the most obvious ones for the



case where the result site (site 3) is different from all the sites that contain files involved in the query (sites 1 and 2). However, suppose that the result site is site 2; then we have two simple strategies:

1. Transfer the EMPLOYEE relation to site 2, execute the query, and present the result to the user at site 2. Here, the same number of bytes—1,000,000—must be transferred for both Q and Q'.
2. Transfer the DEPARTMENT relation to site 1, execute the query at site 1, and send the result back to site 2. In this case  $400,000 + 3,500 = 403,500$  bytes must be transferred for Q and  $4,000 + 3,500 = 7,500$  bytes for Q'.

A more complex strategy, which sometimes works better than these simple strategies, uses an operation called **semijoin**. We introduce this operation and discuss distributed execution using semijoins next.

### 23.5.3 Distributed Query Processing Using Semijoin

The idea behind distributed query processing using the *semijoin operation* is to reduce the number of tuples in a relation before transferring it to another site. Intuitively, the idea is to send the *joining column* of one relation *R* to the site where the other relation *S* is located; this column is then joined with *S*. Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with *R*. Hence, only the joining column of *R* is transferred in one direction, and a subset of *S* with no extraneous tuples or attributes is transferred in the other direction. If only a small fraction of the tuples in *S* participate in the join, this can be an efficient solution to minimizing data transfer.

To illustrate this, consider the following strategy for executing Q or Q':

1. Project the join attributes of DEPARTMENT at site 2, and transfer them to site 1. For Q, we transfer  $F = \pi_{\text{Dnumber}}(\text{DEPARTMENT})$ , whose size is  $4 \times 100 = 400$  bytes, whereas for Q', we transfer  $F' = \pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$ , whose size is  $9 \times 100 = 900$  bytes.
2. Join the transferred file with the EMPLOYEE relation at site 1, and transfer the required attributes from the resulting file to site 2. For Q, we transfer  $R = \pi_{\text{Dno, Fname, Lname}}(F \bowtie_{\text{Dnumber=Dno}} \text{EMPLOYEE})$ , whose size is  $34 \times 10,000 = 340,000$  bytes, whereas for Q', we transfer  $R' = \pi_{\text{Mgr\_ssn, Fname, Lname}}(F' \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$ , whose size is  $39 \times 100 = 3,900$  bytes.
3. Execute the query by joining the transferred file *R* or *R'* with DEPARTMENT, and present the result to the user at site 2.

Using this strategy, we transfer 340,400 bytes for Q and 4,800 bytes for Q'. We limited the EMPLOYEE attributes and tuples transmitted to site 2 in step 2 to only those that will *actually be joined* with a DEPARTMENT tuple in step 3. For query Q, this turned out to include all EMPLOYEE tuples, so little improvement was achieved. However, for Q' only 100 out of the 10,000 EMPLOYEE tuples were needed.

The semijoin operation was devised to formalize this strategy. A **semijoin operation**  $R \bowtie_{A=B} S$ , where  $A$  and  $B$  are domain-compatible attributes of  $R$  and  $S$ , respectively, produces the same result as the relational algebra expression  $\pi R(R \bowtie_{A=B} S)$ . In a distributed environment where  $R$  and  $S$  reside at different sites, the semijoin is typically implemented by first transferring  $F = \pi_B(S)$  to the site where  $R$  resides and then joining  $F$  with  $R$ , thus leading to the strategy discussed here.

Notice that the semijoin operation is not commutative; that is,

$$R \bowtie S \neq S \bowtie R$$

### 23.5.4 Query and Update Decomposition

In a DDBMS with *no distribution transparency*, the user phrases a query directly in terms of specific fragments. For example, consider another query  $Q$ : *Retrieve the names and hours per week for each employee who works on some project controlled by department 5*, which is specified on the distributed database where the relations at sites 2 and 3 are shown in Figure 23.2, and those at site 1 are shown in Figure 5.6, as in our earlier example. A user who submits such a query must specify whether it references the PROJS\_5 and WORKS\_ON\_5 relations at site 2 (Figure 23.2) or the PROJECT and WORKS\_ON relations at site 1 (Figure 5.6). The user must also maintain consistency of replicated data items when updating a DDBMS with *no replication transparency*.

On the other hand, a DDBMS that supports *full distribution, fragmentation, and replication transparency* allows the user to specify a query or update request on the schema in Figure 5.5 just as though the DBMS were centralized. For updates, the DDBMS is responsible for maintaining *consistency among replicated items* by using one of the distributed concurrency control algorithms discussed in Section 23.3. For queries, a **query decomposition** module must break up or **decompose** a query into **subqueries** that can be executed at the individual sites. Additionally, a strategy for combining the results of the subqueries to form the query result must be generated. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or **materialize** a particular replica during query execution.

To determine which replicas include the data items referenced in a query, the DDBMS refers to the fragmentation, replication, and distribution information stored in the DDBMS catalog. For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition, sometimes called a **guard**, is kept for each fragment. This is basically a selection condition that specifies which tuples exist in the fragment; it is called a guard because *only tuples that satisfy this condition* are permitted to be stored in the fragment. For mixed fragments, both the attribute list and the guard condition are kept in the catalog.

In our earlier example, the guard conditions for fragments at site 1 (Figure 5.6) are TRUE (all tuples), and the attribute lists are \* (all attributes). For the fragments

- (a) EMPD5  
 attribute list: Fname, Minit, Lname, Ssn, Salary, Super\_ssn, Dno  
 guard condition: Dno = 5  
 DEP5  
 attribute list: \* (all attributes Dname, Dnumber, Mgr\_ssn, Mgr\_start\_date)  
 guard condition: Dnumber = 5  
 DEP5\_LOCS  
 attribute list: \* (all attributes Dnumber, Location)  
 guard condition: Dnumber = 5  
 PROJS5  
 attribute list: \* (all attributes Pname, Pnumber, Plocation, Dnum)  
 guard condition: Dnum = 5  
 WORKS\_ON5  
 attribute list: \* (all attributes Essn, Pno, Hours)  
 guard condition: Essn IN ( $\pi_{\text{Ssn}}$  (EMPD5)) OR Pno IN ( $\pi_{\text{Pnumber}}$  (PROJS5))
- (b) EMPD4  
 attribute list: Fname, Minit, Lname, Ssn, Salary, Super\_ssn, Dno  
 guard condition: Dno = 4  
 DEP4  
 attribute list: \* (all attributes Dname, Dnumber, Mgr\_ssn, Mgr\_start\_date)  
 guard condition: Dnumber = 4  
 DEP4\_LOCS  
 attribute list: \* (all attributes Dnumber, Location)  
 guard condition: Dnumber = 4  
 PROJS4  
 attribute list: \* (all attributes Pname, Pnumber, Plocation, Dnum)  
 guard condition: Dnum = 4  
 WORKS\_ON4  
 attribute list: \* (all attributes Essn, Pno, Hours)  
 guard condition: Essn IN ( $\pi_{\text{Ssn}}$  (EMPD4))  
 OR Pno IN ( $\pi_{\text{Pnumber}}$  (PROJS4))

**Figure 23.5**

Guard conditions and attributes lists for fragments.

- (a) Site 2 fragments.  
 (b) Site 3 fragments.

shown in Figure 23.2, we have the guard conditions and attribute lists shown in Figure 23.5. When the DDBMS decomposes an update request, it can determine which fragments must be updated by examining their guard conditions. For example, a user request to insert a new EMPLOYEE tuple  $\langle \text{'Alex'}, \text{'B'}, \text{'Coleman'}, \text{'345671239'}, \text{'22-APR-64'}, \text{'3306 Sandstone, Houston, TX'}, \text{M}, 33000, \text{'987654321'}, 4 \rangle$  would be decomposed by the DDBMS into two insert requests: the first inserts the preceding tuple in the EMPLOYEE fragment at site 1, and the second inserts the projected tuple  $\langle \text{'Alex'}, \text{'B'}, \text{'Coleman'}, \text{'345671239'}, 33000, \text{'987654321'}, 4 \rangle$  in the EMPD4 fragment at site 3.

For query decomposition, the DDBMS can determine which fragments may contain the required tuples by comparing the query condition with the guard conditions. For

example, consider the query Q: *Retrieve the names and hours per week for each employee who works on some project controlled by department 5.* This can be specified in SQL on the schema in Figure 5.5 as follows:

```
Q: SELECT Fname, Lname, Hours
   FROM EMPLOYEE, PROJECT, WORKS_ON
   WHERE Dnum=5 AND Pnumber=Pno AND Essn=Ssn;
```

Suppose that the query is submitted at site 2, which is where the query result will be needed. The DDBMS can determine from the guard condition on PROJS5 and WORKS\_ON5 that all tuples satisfying the conditions ( $Dnum = 5$  AND  $Pnumber = Pno$ ) reside at site 2. Hence, it may decompose the query into the following relational algebra subqueries:

$$T_1 \leftarrow \pi_{Essn}(PROJS5 \bowtie_{Pnumber=Pno} WORKS\_ON5)$$

$$T_2 \leftarrow \pi_{Essn, Fname, Lname}(T_1 \bowtie_{Essn=Ssn} EMPLOYEE)$$

$$RESULT \leftarrow \pi_{Fname, Lname, Hours}(T_2 * WORKS\_ON5)$$

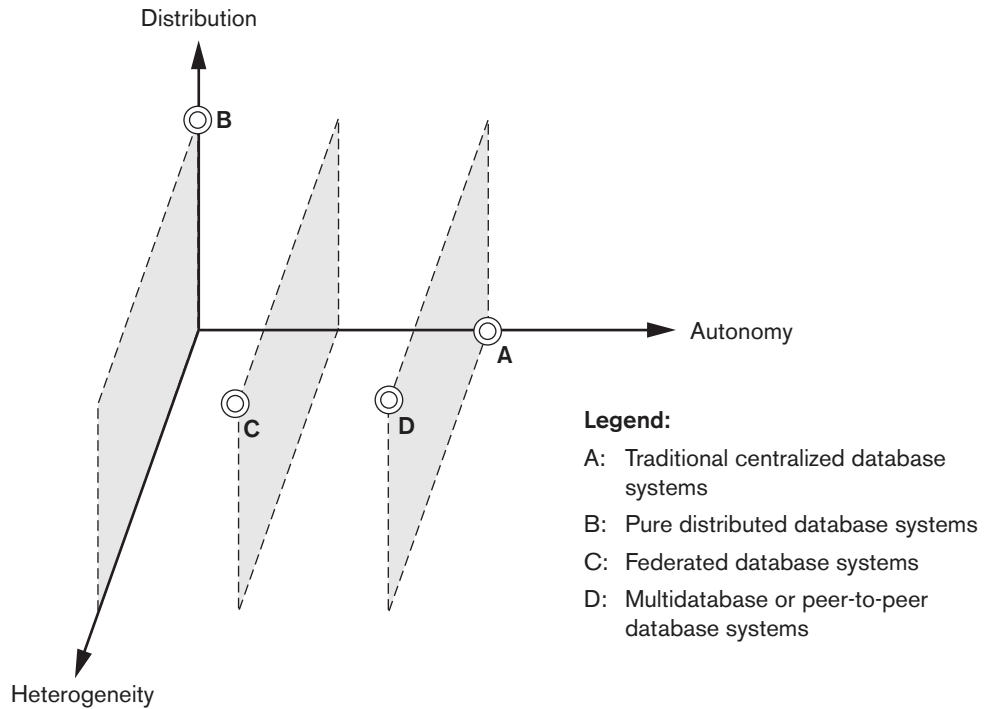
This decomposition can be used to execute the query by using a semijoin strategy. The DDBMS knows from the guard conditions that PROJS5 contains exactly those tuples satisfying ( $Dnum = 5$ ) and that WORKS\_ON5 contains all tuples to be joined with PROJS5; hence, subquery  $T_1$  can be executed at site 2, and the projected column Essn can be sent to site 1. Subquery  $T_2$  can then be executed at site 1, and the result can be sent back to site 2, where the final query result is calculated and displayed to the user. An alternative strategy would be to send the query Q itself to site 1, which includes all the database tuples, where it would be executed locally and from which the result would be sent back to site 2. The query optimizer would estimate the costs of both strategies and would choose the one with the lower cost estimate.

## 23.6 Types of Distributed Database Systems

The term *distributed database management system* can describe various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network. In this section, we discuss a number of types of DDBMSs and the criteria and factors that make some of these systems different.

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all users (clients) use identical software, the DDBMS is called **homogeneous**; otherwise, it is called **heterogeneous**. Another factor related to the degree of homogeneity is the **degree of local autonomy**. If there is no provision for the local site to function as a standalone DBMS, then the system has **no local autonomy**. On the other hand, if *direct access* by local transactions to a server is permitted, the system has some degree of local autonomy.

Figure 23.6 shows classification of DDBMS alternatives along orthogonal axes of distribution, autonomy, and heterogeneity. For a centralized database, there is



complete autonomy but a total lack of distribution and heterogeneity (point A in the figure). We see that the degree of local autonomy provides further ground for classification into federated and multidatabase systems. At one extreme of the autonomy spectrum, we have a DDBMS that *looks like* a centralized DBMS to the user, with zero autonomy (point B). A single conceptual schema exists, and all access to the system is obtained through a site that is part of the DDBMS—which means that no local autonomy exists. Along the autonomy axis we encounter two types of DDBMSs called *federated database system* (point C) and *multidatabase system* (point D). In such systems, each server is an independent and autonomous centralized DBMS that has its own local users, local transactions, and DBA, and hence has a very high degree of *local autonomy*. The term **federated database system (FDBS)** is used when there is some global view or schema of the federation of databases that is shared by the applications (point C). On the other hand, a **multidatabase system** has full local autonomy in that it does not have a global schema but interactively constructs one as needed by the application (point D). Both systems are hybrids between distributed and centralized systems, and the distinction we made between them is not strictly followed. We will refer to them as FDBSs in a generic sense. Point D in the diagram may also stand for a system with full local autonomy and full heterogeneity—this could be a peer-to-peer database system. In a heterogeneous FDBS, one server may be a relational DBMS, another a network DBMS (such as Computer Associates' IDMS or HP'S IMAGE/3000), and

a third an object DBMS (such as Object Design's ObjectStore) or hierarchical DBMS (such as IBM's IMS); in such a case, it is necessary to have a canonical system language and to include language translators to translate subqueries from the canonical language to the language of each server.

We briefly discuss the issues affecting the design of FDBSs next.

### 23.6.1 Federated Database Management Systems Issues

The type of heterogeneity present in FDBSs may arise from several sources. We discuss these sources first and then point out how the different types of autonomies contribute to a semantic heterogeneity that must be resolved in a heterogeneous FDBS.

- **Differences in data models.** Databases in an organization come from a variety of data models, including the so-called legacy models (hierarchical and network), the relational data model, the object data model, and even files. The modeling capabilities of the models vary. Hence, to deal with them uniformly via a single global schema or to process them in a single language is challenging. Even if two databases are both from the RDBMS environment, the same information may be represented as an attribute name, as a relation name, or as a value in different databases. This calls for an intelligent query-processing mechanism that can relate information based on metadata.
- **Differences in constraints.** Constraint facilities for specification and implementation vary from system to system. There are comparable features that must be reconciled in the construction of a global schema. For example, the relationships from ER models are represented as referential integrity constraints in the relational model. Triggers may have to be used to implement certain constraints in the relational model. The global schema must also deal with potential conflicts among constraints.
- **Differences in query languages.** Even with the same data model, the languages and their versions vary. For example, SQL has multiple versions like SQL-89, SQL-92, SQL-99, and SQL:2008, and each system has its own set of data types, comparison operators, string manipulation features, and so on.

**Semantic Heterogeneity.** Semantic heterogeneity occurs when there are differences in the meaning, interpretation, and intended use of the same or related data. Semantic heterogeneity among component database systems (DBSs) creates the biggest hurdle in designing global schemas of heterogeneous databases. The **design autonomy** of component DBSs refers to their freedom of choosing the following design parameters; the design parameters in turn affect the eventual complexity of the FDBS:

- **The universe of discourse from which the data is drawn.** For example, for two customer accounts, databases in the federation may be from the United States and Japan and have entirely different sets of attributes about customer accounts required by the accounting practices. Currency rate fluctuations

would also present a problem. Hence, relations in these two databases that have identical names—CUSTOMER or ACCOUNT—may have some common and some entirely distinct information.

- **Representation and naming.** The representation and naming of data elements and the structure of the data model may be prespecified for each local database.
- **The understanding, meaning, and subjective interpretation of data.** This is a chief contributor to semantic heterogeneity.
- **Transaction and policy constraints.** These deal with serializability criteria, compensating transactions, and other transaction policies.
- **Derivation of summaries.** Aggregation, summarization, and other data-processing features and operations supported by the system.

The above problems related to semantic heterogeneity are being faced by all major multinational and governmental organizations in all application areas. In today's commercial environment, most enterprises are resorting to heterogeneous FDBSs, having heavily invested in the development of individual database systems using diverse data models on different platforms over the last 20 to 30 years. Enterprises are using various forms of software—typically called the **middleware**; or Web-based packages called **application servers** (for example, WebLogic or WebSphere); and even generic systems, called **enterprise resource planning (ERP) systems** (for example, SAP, J. D. Edwards ERP)—to manage the transport of queries and transactions from the global application to individual databases (with possible additional processing for business rules) and the data from the heterogeneous database servers to the global application. Detailed discussion of these types of software systems is outside the scope of this text.

Just as providing the ultimate transparency is the goal of any distributed database architecture, local component databases strive to preserve autonomy. **Communication autonomy** of a component DBS refers to its ability to decide whether to communicate with another component DBS. **Execution autonomy** refers to the ability of a component DBS to execute local operations without interference from external operations by other component DBSs and its ability to decide the order in which to execute them. The **association autonomy** of a component DBS implies that it has the ability to decide whether and how much to share its functionality (operations it supports) and resources (data it manages) with other component DBSs. The major challenge of designing FDBSs is to let component DBSs interoperate while still providing the above types of autonomies to them.

## 23.7 Distributed Database Architectures

In this section, we first briefly point out the distinction between parallel and distributed database architectures. Although both are prevalent in industry today, there are various manifestations of the distributed architectures that are continuously evolving among large enterprises. The parallel architecture is more common in high-per-



formance computing, where there is a need for multiprocessor architectures to cope with the volume of data undergoing transaction processing and warehousing applications. We then introduce a generic architecture of a distributed database. This is followed by discussions on the architecture of three-tier client/server and federated database systems.

### 23.7.1 Parallel versus Distributed Architectures

There are two main types of multiprocessor system architectures that are commonplace:

- **Shared memory (tightly coupled) architecture.** Multiple processors share secondary (disk) storage and also share primary memory.
- **Shared disk (loosely coupled) architecture.** Multiple processors share secondary (disk) storage but each has their own primary memory.

These architectures enable processors to communicate without the overhead of exchanging messages over a network.<sup>4</sup> Database management systems developed using the above types of architectures are termed **parallel database management systems** rather than DDBMSs, since they utilize parallel processor technology. Another type of multiprocessor architecture is called **shared-nothing architecture**. In this architecture, every processor has its own primary and secondary (disk) memory, no common memory exists, and the processors communicate over a high-speed interconnection network (bus or switch). Although the shared-nothing architecture resembles a distributed database computing environment, major differences exist in the mode of operation. In shared-nothing multiprocessor systems, there is symmetry and homogeneity of nodes; this is not true of the distributed database environment, where heterogeneity of hardware and operating system at each node is very common. Shared-nothing architecture is also considered as an environment for parallel databases. Figure 23.7(a) illustrates a parallel database (shared nothing), whereas Figure 23.7(b) illustrates a centralized database with distributed access and Figure 23.7(c) shows a pure distributed database. We will not expand on parallel architectures and related data management issues here.

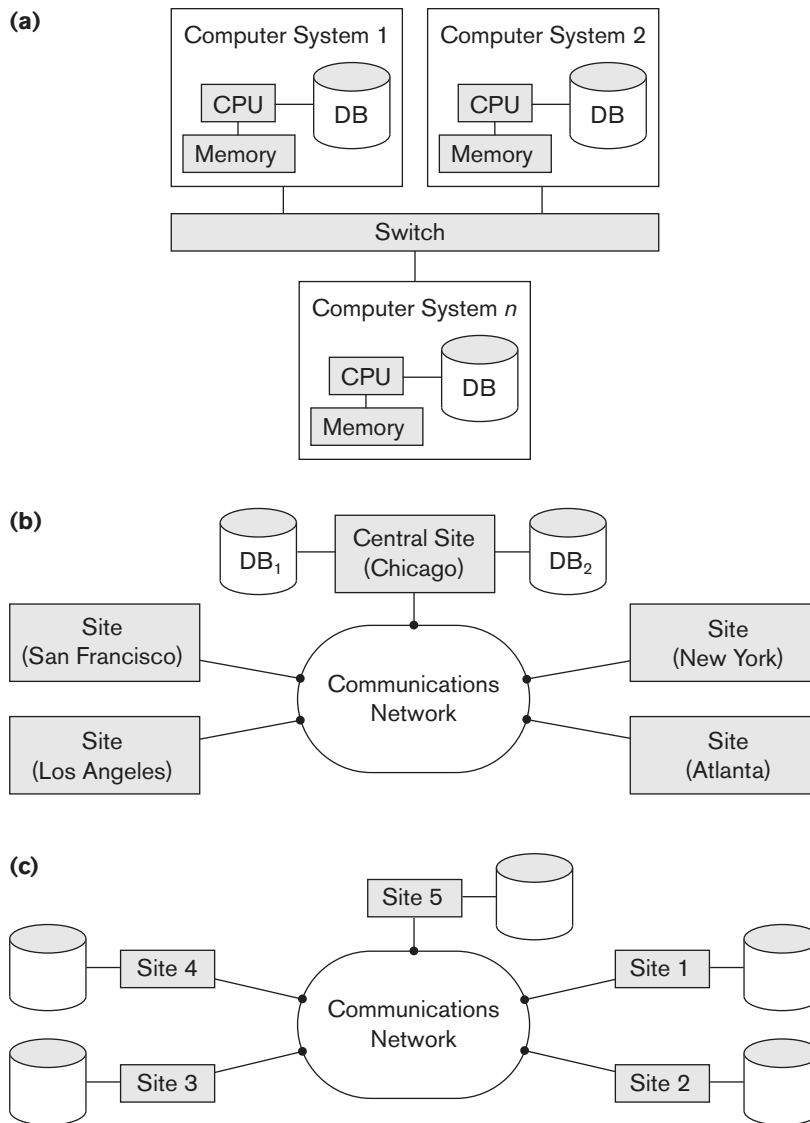
### 23.7.2 General Architecture of Pure Distributed Databases

In this section, we discuss both the logical and component architectural models of a DDB. In Figure 23.8, which describes the generic schema architecture of a DDB, the enterprise is presented with a consistent, unified view showing the logical structure of underlying data across all nodes. This view is represented by the global conceptual schema (GCS), which provides network transparency (see Section 23.1.2). To accommodate potential heterogeneity in the DDB, each node is shown as having its own local internal schema (LIS) based on physical organization details at that

---

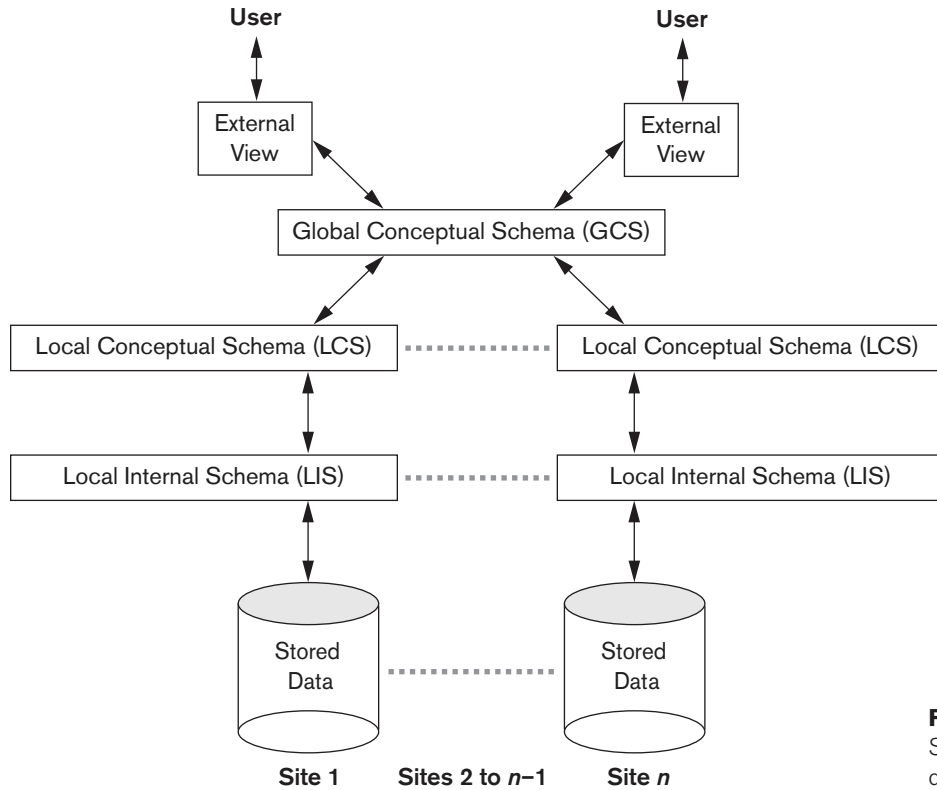
<sup>4</sup>If both primary and secondary memories are shared, the architecture is also known as *shared-everything architecture*.



**Figure 23.7**

Some different database system architectures. (a) Shared-nothing architecture. (b) A networked architecture with a centralized database at one of the sites. (c) A truly distributed database architecture.

particular site. The logical organization of data at each site is specified by the local conceptual schema (LCS). The GCS, LCS, and their underlying mappings provide the fragmentation and replication transparency discussed in Section 23.1.2. Figure 23.8 shows the component architecture of a DDB. It is an extension of its centralized counterpart (Figure 2.3) in Chapter 2. For the sake of simplicity, common

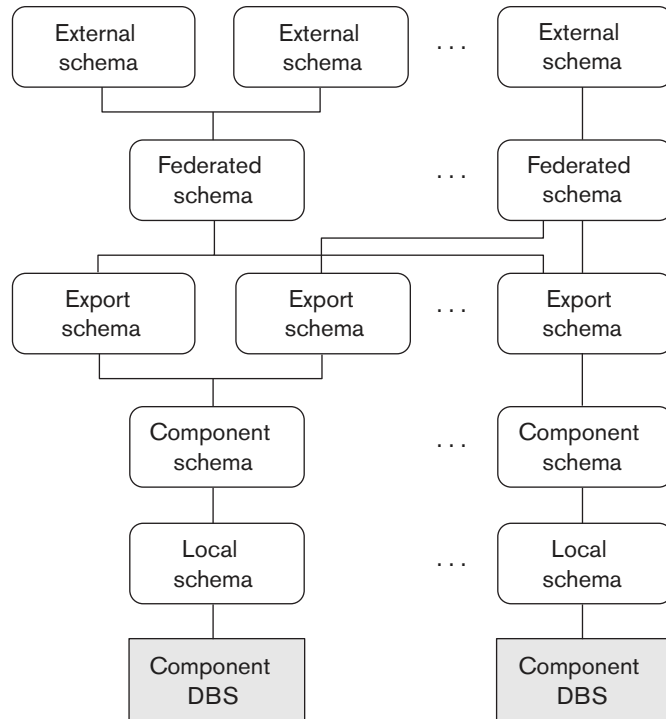


**Figure 23.8**  
Schema architecture of  
distributed databases.

elements are not shown here. The global query compiler references the global conceptual schema from the global system catalog to verify and impose defined constraints. The global query optimizer references both global and local conceptual schemas and generates optimized local queries from global queries. It evaluates all candidate strategies using a cost function that estimates cost based on response time (CPU, I/O, and network latencies) and estimated sizes of intermediate results. The latter is particularly important in queries involving joins. Having computed the cost for each candidate, the optimizer selects the candidate with the minimum cost for execution. Each local DBMS would have its local query optimizer, transaction manager, and execution engines as well as the local system catalog, which houses the local schemas. The global transaction manager is responsible for coordinating the execution across multiple sites in conjunction with the local transaction manager at those sites.

### 23.7.3 Federated Database Schema Architecture

Typical five-level schema architecture to support global applications in the FDBS environment is shown in Figure 23.9. In this architecture, the **local schema** is the

**Figure 23.9**

The five-level schema architecture in a federated database system (FDBS).

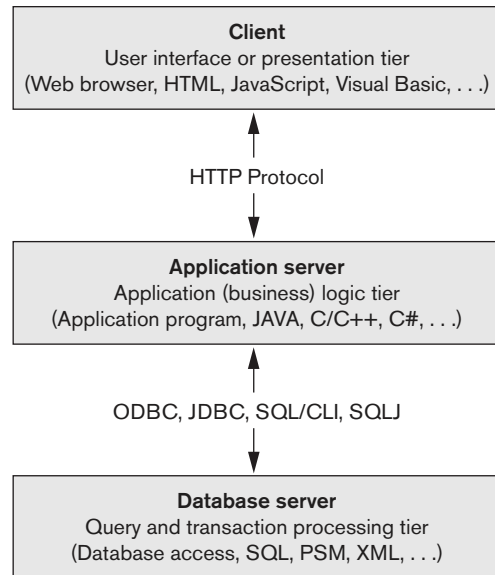
Source: Adapted from Sheth and Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Computing Surveys* (Vol. 22: No. 3, September 1990).

conceptual schema (full database definition) of a component database, and the **component schema** is derived by translating the local schema into a canonical data model or common data model (CDM) for the FDBS. Schema translation from the local schema to the component schema is accompanied by generating mappings to transform commands on a component schema into commands on the corresponding local schema. The **export schema** represents the subset of a component schema that is available to the FDBS. The **federated schema** is the global schema or view, which is the result of integrating all the shareable export schemas. The **external schemas** define the schema for a user group or an application, as in the three-level schema architecture.

All the problems related to query processing, transaction processing, and directory and metadata management and recovery apply to FDBSs with additional considerations. It is not within our scope to discuss them in detail here.

### 23.7.4 An Overview of Three-Tier Client/Server Architecture

As we pointed out in the chapter introduction, full-scale DDBMSs have not been developed to support all the types of functionalities that we have discussed so far. Instead, distributed database applications are being developed in the context of the client/server architectures. We introduced the two-tier client/server architecture in

**Figure 23.10**

The three-tier client/server architecture.

Section 2.5. It is now more common to use a three-tier architecture rather than a two-tier architecture, particularly in Web applications. This architecture is illustrated in Figure 23.10.

In the three-tier client/server architecture, the following three layers exist:

1. **Presentation layer (client).** This provides the user interface and interacts with the user. The programs at this layer present Web interfaces or forms to the client in order to interface with the application. Web browsers are often utilized, and the languages and specifications used include HTML, XHTML, CSS, Flash, MathML, Scalable Vector Graphics (SVG), Java, JavaScript, Adobe Flex, and others. This layer handles user input, output, and navigation by accepting user commands and displaying the needed information, usually in the form of static or dynamic Web pages. The latter are employed when the interaction involves database access. When a Web interface is used, this layer typically communicates with the application layer via the HTTP protocol.
2. **Application layer (business logic).** This layer programs the application logic. For example, queries can be formulated based on user input from the client, or query results can be formatted and sent to the client for presentation. Additional application functionality can be handled at this layer, such as security checks, identity verification, and other functions. The application layer can interact with one or more databases or data sources as needed by connecting to the database using ODBC, JDBC, SQL/CLI, or other database access techniques.

3. **Database server.** This layer handles query and update requests from the application layer, processes the requests, and sends the results. Usually SQL is used to access the database if it is relational or object-relational, and stored database procedures may also be invoked. Query results (and queries) may be formatted into XML (see Chapter 13) when transmitted between the application server and the database server.

Exactly how to divide the DBMS functionality among the client, application server, and database server may vary. The common approach is to include the functionality of a centralized DBMS at the database server level. A number of relational DBMS products have taken this approach, in which an **SQL server** is provided. The application server must then formulate the appropriate SQL queries and connect to the database server when needed. The client provides the processing for user interface interactions. Since SQL is a relational standard, various SQL servers, possibly provided by different vendors, can accept SQL commands through standards such as ODBC, JDBC, and SQL/CLI (see Chapter 10).

In this architecture, the application server may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between an application server and database server might proceed as follows during the processing of an SQL query:

1. The application server formulates a user query based on input from the client layer and decomposes it into a number of independent site queries. Each site query is sent to the appropriate database server site.
2. Each database server processes the local query and sends the results to the application server site. Increasingly, XML is being touted as the standard for data exchange (see Chapter 13), so the database server may format the query result into XML before sending it to the application server.
3. The application server combines the results of the subqueries to produce the result of the originally required query, formats it into HTML or some other form accepted by the client, and sends it to the client site for display.

The application server is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transactions to be executed, as well as commands to transmit data to other clients or servers. Another function controlled by the application server (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The application server must also ensure the atomicity of global transactions by performing global recovery when certain sites fail.

If the DDBMS has the capability to *hide* the details of data distribution from the application server, then it enables the application server to execute global queries and transactions as though the database were centralized, without having to specify

the sites at which the data referenced in the query or transaction resides. This property is called **distribution transparency**. Some DDBMSs do not provide distribution transparency, instead requiring that applications are aware of the details of data distribution.

## 23.8 Distributed Catalog Management

Efficient catalog management in distributed databases is critical to ensure satisfactory performance related to site autonomy, view management, and data distribution and replication. Catalogs are databases themselves containing metadata about the distributed database system.

Three popular management schemes for distributed catalogs are *centralized* catalogs, *fully replicated* catalogs, and *partitioned* catalogs. The choice of the scheme depends on the database itself as well as the access patterns of the applications to the underlying data.

**Centralized Catalogs.** In this scheme, the entire catalog is stored in one single site. Due to its central nature, it is easy to implement. On the other hand, the advantages of reliability, availability, autonomy, and distribution of processing load are adversely impacted. For read operations from noncentral sites, the requested catalog data is locked at the central site and is then sent to the requesting site. On completion of the read operation, an acknowledgment is sent to the central site, which in turn unlocks this data. All update operations must be processed through the central site. This can quickly become a performance bottleneck for write-intensive applications.

**Fully Replicated Catalogs.** In this scheme, identical copies of the complete catalog are present at each site. This scheme facilitates faster reads by allowing them to be answered locally. However, all updates must be broadcast to all sites. Updates are treated as transactions, and a centralized two-phase commit scheme is employed to ensure catalog consistency. As with the centralized scheme, write-intensive applications may cause increased network traffic due to the broadcast associated with the writes.

**Partially Replicated Catalogs.** The centralized and fully replicated schemes restrict site autonomy since they must ensure a consistent global view of the catalog. Under the partially replicated scheme, each site maintains complete catalog information on data stored locally at that site. Each site is also permitted to cache entries retrieved from remote sites. However, there are no guarantees that these cached copies will be the most recent and updated. The system tracks catalog entries for sites where the object was created and for sites that contain copies of this object. Any changes to copies are propagated immediately to the original (birth) site. Retrieving updated copies to replace stale data may be delayed until an access to this data occurs. In general, fragments of relations across sites should be uniquely accessible. Also, to ensure data distribution transparency, users should be allowed to create synonyms for remote objects and use these synonyms for subsequent referrals.

## 23.9 Summary

In this chapter, we provided an introduction to distributed databases. This is a very broad topic, and we discussed only some of the basic techniques used with distributed databases. First in Section 23.1 we discussed the reasons for distribution and DDB concepts in Section 23.1.1. Then we defined the concept of distribution transparency and the related concepts of fragmentation transparency and replication transparency in Section 23.1.2. We discussed the concepts of distributed availability and reliability in Section 23.1.3, and gave an overview of scalability and partition tolerance issues in Section 23.1.4. We discussed autonomy of nodes in a distributed system in Section 23.1.5 and the potential advantages of distributed databases over centralized system in Section 23.1.6.

In Section 23.2, we discussed the design issues related to data fragmentation, replication, and distribution. We distinguished between horizontal fragmentation (sharding) and vertical fragmentation of relations in Section 23.2.1. We then discussed in Section 23.2.2 the use of data replication to improve system reliability and availability. In Section 23.3, we briefly discussed the concurrency control and recovery techniques used in DDBMSs, and then reviewed some of the additional problems that must be dealt with in a distributed environment that do not appear in a centralized environment. Then in Section 23.4 we discussed transaction management, including different commit protocols (2-phase commit, 3-phase commit) and operating system support for transaction management.

We then illustrated some of the techniques used in distributed query processing in Section 23.5, and discussed the cost of communication among sites, which is considered a major factor in distributed query optimization. We compared the different techniques for executing joins, and we then presented the semijoin technique for joining relations that reside on different sites in Section 23.5.3.

Following that, in Section 23.6, we categorized DDBMSs by using criteria such as the degree of homogeneity of software modules and the degree of local autonomy. In Section 23.7 we distinguished between parallel and distributed system architectures and then introduced the generic architecture of distributed databases from both a component as well as a schematic architectural perspective. In Section 23.7.3 we discussed in some detail issues of federated database management, and we focused on the needs of supporting various types of autonomies and dealing with semantic heterogeneity. We also reviewed the client/server architecture concepts and related them to distributed databases in Section 23.7.4. We reviewed catalog management in distributed databases and summarized their relative advantages and disadvantages in Section 23.8.

Chapters 24 and 25 will describe recent advances in distributed databases and distributed computing related to big data. Chapter 24 describes the so-called NOSQL systems, which are highly scalable, distributed database systems that handle large volumes of data. Chapter 25 discusses cloud computing and distributed computing technologies that are needed to process big data.

## Review Questions

- 23.1. What are the main reasons for and potential advantages of distributed databases?
- 23.2. What additional functions does a DDBMS have over a centralized DBMS?
- 23.3. Discuss what is meant by the following terms: *degree of homogeneity of a DDBMS*, *degree of local autonomy of a DDBMS*, *federated DBMS*, *distribution transparency*, *fragmentation transparency*, *replication transparency*, *multidatabase system*.
- 23.4. Discuss the architecture of a DDBMS. Within the context of a centralized DBMS, briefly explain new components introduced by the distribution of data.
- 23.5. What are the main software modules of a DDBMS? Discuss the main functions of each of these modules in the context of the client/server architecture.
- 23.6. Compare the two-tier and three-tier client/server architectures.
- 23.7. What is a fragment of a relation? What are the main types of fragments? Why is fragmentation a useful concept in distributed database design?
- 23.8. Why is data replication useful in DDBMSs? What typical units of data are replicated?
- 23.9. What is meant by *data allocation* in distributed database design? What typical units of data are distributed over sites?
- 23.10. How is a horizontal partitioning of a relation specified? How can a relation be put back together from a complete horizontal partitioning?
- 23.11. How is a vertical partitioning of a relation specified? How can a relation be put back together from a complete vertical partitioning?
- 23.12. Discuss the naming problem in distributed databases.
- 23.13. What are the different stages of processing a query in a DDBMS?
- 23.14. Discuss the different techniques for executing an equijoin of two files located at different sites. What main factors affect the cost of data transfer?
- 23.15. Discuss the semijoin method for executing an equijoin of two files located at different sites. Under what conditions is an equijoin strategy efficient?
- 23.16. Discuss the factors that affect query decomposition. How are guard conditions and attribute lists of fragments used during the query decomposition process?
- 23.17. How is the decomposition of an update request different from the decomposition of a query? How are guard conditions and attribute lists of fragments used during the decomposition of an update request?



- 23.18.** List the support offered by operating systems to a DDBMS and also the benefits of these supports.
- 23.19.** Discuss the factors that do not appear in centralized systems but that affect concurrency control and recovery in distributed systems.
- 23.20.** Discuss the two-phase commit protocol used for transaction management in a DDBMS. List its limitations and explain how they are overcome using the three-phase commit protocol.
- 23.21.** Compare the primary site method with the primary copy method for distributed concurrency control. How does the use of backup sites affect each?
- 23.22.** When are voting and elections used in distributed databases?
- 23.23.** Discuss catalog management in distributed databases.
- 23.24.** What are the main challenges facing a traditional DDBMS in the context of today's Internet applications? How does cloud computing attempt to address them?
- 23.25.** Discuss briefly the support offered by Oracle for homogeneous, heterogeneous, and client/server-based distributed database architectures.
- 23.26.** Discuss briefly online directories, their management, and their role in distributed databases.

## Exercises

- 23.27.** Consider the data distribution of the COMPANY database, where the fragments at sites 2 and 3 are as shown in Figure 23.3 and the fragments at site 1 are as shown in Figure 3.6. For each of the following queries, show at least two strategies of decomposing and executing the query. Under what conditions would each of your strategies work well?
  - a. For each employee in department 5, retrieve the employee name and the names of the employee's dependents.
  - b. Print the names of all employees who work in department 5 but who work on some project *not* controlled by department 5.
- 23.28.** Consider the following relations:

BOOKS(Book#, Primary\_author, Topic, Total\_stock, \$price)  
 BOOKSTORE(Store#, City, State, Zip, Inventory\_value)  
 STOCK(Store#, Book#, Qty)

Total\_stock is the total number of books in stock, and Inventory\_value is the total inventory value for the store in dollars.

- a. Give an example of two simple predicates that would be meaningful for the BOOKSTORE relation for horizontal partitioning.

- b. How would a derived horizontal partitioning of STOCK be defined based on the partitioning of BOOKSTORE?
- c. Show predicates by which BOOKS may be horizontally partitioned by topic.
- d. Show how the STOCK may be further partitioned from the partitions in (b) by adding the predicates in (c).

**23.29.** Consider a distributed database for a bookstore chain called National Books with three sites called EAST, MIDDLE, and WEST. The relation schemas are given in Exercise 23.28. Consider that BOOKS are fragmented by \$price amounts into:

$B_1$ : BOOK1: \$price up to \$20  
 $B_2$ : BOOK2: \$price from \$20.01 to \$50  
 $B_3$ : BOOK3: \$price from \$50.01 to \$100  
 $B_4$ : BOOK4: \$price \$100.01 and above

Similarly, BOOK\_STORES are divided by zip codes into:

$S_1$ : EAST: Zip up to 35000  
 $S_2$ : MIDDLE: Zip 35001 to 70000  
 $S_3$ : WEST: Zip 70001 to 99999

Assume that STOCK is a derived fragment based on BOOKSTORE only.

- a. Consider the query:

```
SELECT  Book#, Total_stock
FROM    Books
WHERE   $price > 15 AND $price < 55;
```

Assume that fragments of BOOKSTORE are nonreplicated and assigned based on region. Assume further that BOOKS are allocated as:

EAST:  $B_1, B_4$   
MIDDLE:  $B_1, B_2$   
WEST:  $B_1, B_2, B_3, B_4$

Assuming the query was submitted in EAST, what remote subqueries does it generate? (Write in SQL.)

- b. If the price of Book# = 1234 is updated from \$45 to \$55 at site MIDDLE, what updates does that generate? Write in English and then in SQL.
- c. Give a sample query issued at WEST that will generate a subquery for MIDDLE.
- d. Write a query involving selection and projection on the above relations and show two possible query trees that denote different ways of execution.

**23.70.** Consider that you have been asked to propose a database architecture in a large organization (General Motors, for example) to consolidate all data

including legacy databases (from hierarchical and network models; no specific knowledge of these models is needed) as well as relational databases, which are geographically distributed so that global applications can be supported. Assume that alternative 1 is to keep all databases as they are, whereas alternative 2 is to first convert them to relational and then support the applications over a distributed integrated database.

- a. Draw two schematic diagrams for the above alternatives showing the linkages among appropriate schemas. For alternative 1, choose the approach of providing export schemas for each database and constructing unified schemas for each application.
- b. List the steps that you would have to go through under each alternative from the present situation until global applications are viable.
- c. Compare these alternatives from the issues of:
  - i. design time considerations
  - ii. runtime considerations

## Selected Bibliography

The textbooks by Ceri and Pelagatti (1984a) and Ozsu and Valduriez (1999) are devoted to distributed databases. Peterson and Davie (2008), Tannenbaum (2003), and Stallings (2007) cover data communications and computer networks. Comer (2008) discusses networks and internets. Ozsu et al. (1994) has a collection of papers on distributed object management.

Most of the research on distributed database design, query processing, and optimization occurred in the 1980s and 1990s; we quickly review the important references here. Distributed database design has been addressed in terms of horizontal and vertical fragmentation, allocation, and replication. Ceri et al. (1982) defined the concept of minterm horizontal fragments. Ceri et al. (1983) developed an integer programming-based optimization model for horizontal fragmentation and allocation. Navathe et al. (1984) developed algorithms for vertical fragmentation based on attribute affinity and showed a variety of contexts for vertical fragment allocation. Wilson and Navathe (1986) present an analytical model for optimal allocation of fragments. Elmasri et al. (1987) discuss fragmentation for the ECR model; Karlapalem et al. (1996) discuss issues for distributed design of object databases. Navathe et al. (1996) discuss mixed fragmentation by combining horizontal and vertical fragmentation; Karlapalem et al. (1996) present a model for redesign of distributed databases.

Distributed query processing, optimization, and decomposition are discussed in Hevner and Yao (1979), Kerschberg et al. (1982), Apers et al. (1983), Ceri and Pelagatti (1984), and Bodorick et al. (1992). Bernstein and Goodman (1981) discuss the theory behind semijoin processing. Wong (1983) discusses the use of relationships in relation fragmentation. Concurrency control and recovery schemes are discussed in Bernstein and Goodman (1981a). Kumar and Hsu (1998) compile some articles

related to recovery in distributed databases. Elections in distributed systems are discussed in Garcia-Molina (1982). Lamport (1978) discusses problems with generating unique timestamps in a distributed system. Rahimi and Haug (2007) discuss a more flexible way to construct query critical metadata for P2P databases. Ouzzani and Bouguettaya (2004) outline fundamental problems in distributed query processing over Web-based data sources.

A concurrency control technique for replicated data that is based on voting is presented by Thomas (1979). Gifford (1979) proposes the use of weighted voting, and Paris (1986) describes a method called *voting with witnesses*. Jajodia and Mutchler (1990) discuss dynamic voting. A technique called *available copy* is proposed by Bernstein and Goodman (1984), and one that uses the idea of a group is presented in ElAbbadi and Toueg (1988). Other work that discusses replicated data includes Gladney (1989), Agrawal and ElAbbadi (1990), ElAbbadi and Toueg (1989), Kumar and Segev (1993), Mukkamala (1989), and Wolfson and Milo (1991). Bassiouni (1988) discusses optimistic protocols for DDB concurrency control. Garcia-Molina (1983) and Kumar and Stonebraker (1987) discuss techniques that use the semantics of the transactions. Distributed concurrency control techniques based on locking and distinguished copies are presented by Menasce et al. (1980) and Minoura and Wiederhold (1982). Obermark (1982) presents algorithms for distributed deadlock detection. In more recent work, Vadivelu et al. (2008) propose using backup mechanism and multilevel security to develop algorithms for improving concurrency. Madria et al. (2007) propose a mechanism based on a multiversion two-phase locking scheme and timestamping to address concurrency issues specific to mobile database systems. Boukerche and Tuck (2001) propose a technique that allows transactions to be out of order to a limited extent. They attempt to ease the load on the application developer by exploiting the network environment and producing a schedule equivalent to a temporally ordered serial schedule. Han et al. (2004) propose a deadlock-free and serializable extended Petri net model for Web-based distributed real-time databases.

A survey of recovery techniques in distributed systems is given by Kohler (1981). Reed (1983) discusses atomic actions on distributed data. Bhargava (1987) presents an edited compilation of various approaches and techniques for concurrency and reliability in distributed systems.

Federated database systems were first defined in McLeod and Heimbigner (1985). Techniques for schema integration in federated databases are presented by Elmasri et al. (1986), Batini et al. (1987), Hayne and Ram (1990), and Motro (1987). Elmagarmid and Helal (1988) and Gamal-Eldin et al. (1988) discuss the update problem in heterogeneous DDBSs. Heterogeneous distributed database issues are discussed in Hsiao and Kamel (1989). Sheth and Larson (1990) present an exhaustive survey of federated database management.

Since the late 1980s, multidatabase systems and interoperability have become important topics. Techniques for dealing with semantic incompatibilities among multiple databases are examined in DeMichiel (1989), Siegel and Madnick (1991), Krishnamurthy et al. (1991), and Wang and Madnick (1989). Castano et al. (1998)

present an excellent survey of techniques for analysis of schemas. Pitoura et al. (1995) discuss object orientation in multidatabase systems. Xiao et al. (2003) propose an XML-based model for a common data model for multidatabase systems and present a new approach for schema mapping based on this model. Lakshmanan et al. (2001) propose extending SQL for interoperability and describe the architecture and algorithms for achieving the same.

Transaction processing in multidatabases is discussed in Mehrotra et al. (1992), Georgakopoulos et al. (1991), Elmagarmid et al. (1990), and Brietbart et al. (1990), among others. Elmagarmid (1992) discusses transaction processing for advanced applications, including engineering applications that are discussed in Heiler et al. (1992).

The workflow systems, which are becoming popular for managing information in complex organizations, use multilevel and nested transactions in conjunction with distributed databases. Weikum (1991) discusses multilevel transaction management. Alonso et al. (1997) discuss limitations of current workflow systems. Lopes et al. (2009) propose that users define and execute their own workflows using a client-side Web browser. They attempt to leverage Web 2.0 trends to simplify the user's work for workflow management. Jung and Yeom (2008) exploit data workflow to develop an improved transaction management system that provides simultaneous, transparent access to the heterogeneous storages that constitute the HVEM DataGrid. Deelman and Chervanek (2008) list the challenges in data-intensive scientific workflows. Specifically, they look at automated management of data, efficient mapping techniques, and user feedback issues in workflow mapping. They also argue for data reuse as an efficient means to manage data and present the challenges therein.

A number of experimental distributed DBMSs have been implemented. These include distributed INGRES by Epstein et al. (1978), DDTS by Devor and Weeldreyer (1980), SDD-1 by Rothnie et al. (1980), System R\* by Lindsay et al. (1984), SIRIUS-DELTA by Ferrier and Stangret (1982), and MULTIBASE by Smith et al. (1981). The OMNIBASE system by Rusinkiewicz et al. (1988) and the Federated Information Base developed using the Candide data model by Navathe et al. (1994) are examples of federated DDBMSs. Pitoura et al. (1995) present a comparative survey of the federated database system prototypes. Most commercial DBMS vendors have products using the client/server approach and offer distributed versions of their systems. Some system issues concerning client/server DBMS architectures are discussed in Carey et al. (1991), DeWitt et al. (1990), and Wang and Rowe (1991). Khoshafian et al. (1992) discuss design issues for relational DBMSs in the client/server environment. Client/server management issues are discussed in many books, such as Zantinge and Adriaans (1996). Di Stefano (2005) discusses data distribution issues specific to grid computing. A major part of this discussion may also apply to cloud computing.

## NOSQL Databases and Big Data Storage Systems

We now turn our attention to the class of systems developed to manage large amounts of data in organizations such as Google, Amazon, Facebook, and Twitter and in applications such as social media, Web links, user profiles, marketing and sales, posts and tweets, road maps and spatial data, and e-mail. The term **NOSQL** is generally interpreted as Not Only SQL—rather than NO to SQL—and is meant to convey that many applications need systems other than traditional relational SQL systems to augment their data management needs. Most NOSQL systems are distributed databases or distributed storage systems, with a focus on semistructured data storage, high performance, availability, data replication, and scalability as opposed to an emphasis on immediate data consistency, powerful query languages, and structured data storage.

We start in Section 24.1 with an introduction to NOSQL systems, their characteristics, and how they differ from SQL systems. We also describe four general categories of NOSQL systems—document-based, key-value stores, column-based, and graph-based. Section 24.2 discusses how NOSQL systems approach the issue of consistency among multiple replicas (copies) by using the paradigm known as **eventual consistency**. We discuss the **CAP** theorem, which can be used to understand the emphasis of NOSQL systems on availability. In Sections 24.3 through 24.6, we present an overview of each category of NOSQL systems—starting with document-based systems, followed by key-value stores, then column-based, and finally graph-based. Some systems may not fall neatly into a single category, but rather use techniques that span two or more categories of NOSQL systems. Finally, Section 24.7 is the chapter summary.

## 24.1 Introduction to NOSQL Systems

### 24.1.1 Emergence of NOSQL Systems

Many companies and organizations are faced with applications that store vast amounts of data. Consider a free e-mail application, such as Google Mail or Yahoo Mail or other similar service—this application can have millions of users, and each user can have thousands of e-mail messages. There is a need for a storage system that can manage all these e-mails; a structured relational SQL system may not be appropriate because (1) SQL systems offer too many services (powerful query language, concurrency control, etc.), which this application may not need; and (2) a structured data model such the traditional relational model may be too restrictive. Although newer relational systems do have more complex object-relational modeling options (see Chapter 12), they still require schemas, which are not required by many of the NOSQL systems.

As another example, consider an application such as Facebook, with millions of users who submit posts, many with images and videos; then these posts must be displayed on pages of other users using the social media relationships among the users. User profiles, user relationships, and posts must all be stored in a huge collection of data stores, and the appropriate posts must be made available to the sets of users that have signed up to see these posts. Some of the data for this type of application is not suitable for a traditional relational system and typically needs multiple types of databases and data storage systems.

Some of the organizations that were faced with these data management and storage applications decided to develop their own systems:

- Google developed a proprietary NOSQL system known as **BigTable**, which is used in many of Google's applications that require vast amounts of data storage, such as Gmail, Google Maps, and Web site indexing. Apache Hbase is an open source NOSQL system based on similar concepts. Google's innovation led to the category of NOSQL systems known as **column-based** or **wide column** stores; they are also sometimes referred to as **column family** stores.
- Amazon developed a NOSQL system called **DynamoDB** that is available through Amazon's cloud services. This innovation led to the category known as **key-value** data stores or sometimes **key-tuple** or **key-object** data stores.
- Facebook developed a NOSQL system called **Cassandra**, which is now open source and known as Apache Cassandra. This NOSQL system uses concepts from both key-value stores and column-based systems.
- Other software companies started developing their own solutions and making them available to users who need these capabilities—for example, **MongoDB** and **CouchDB**, which are classified as **document-based** NOSQL systems or **document stores**.
- Another category of NOSQL systems is the **graph-based** NOSQL systems, or **graph databases**; these include **Neo4J** and **GraphBase**, among others.



- Some NOSQL systems, such as **OrientDB**, combine concepts from many of the categories discussed above.
- In addition to the newer types of NOSQL systems listed above, it is also possible to classify database systems based on the object model (see Chapter 12) or on the native XML model (see Chapter 13) as NOSQL systems, although they may not have the high-performance and replication characteristics of the other types of NOSQL systems.

These are just a few examples of NOSQL systems that have been developed. There are many systems, and listing all of them is beyond the scope of our presentation.

### 24.1.2 Characteristics of NOSQL Systems

We now discuss the characteristics of many NOSQL systems, and how these systems differ from traditional SQL systems. We divide the characteristics into two categories—those related to distributed databases and distributed systems, and those related to data models and query languages.

**NOSQL characteristics related to distributed databases and distributed systems.** NOSQL systems emphasize high availability, so replicating the data is inherent in many of these systems. Scalability is another important characteristic, because many of the applications that use NOSQL systems tend to have data that keeps growing in volume. High performance is another required characteristic, whereas serializable consistency may not be as important for some of the NOSQL applications. We discuss some of these characteristics next.

1. **Scalability:** As we discussed in Section 23.1.4, there are two kinds of scalability in distributed systems: horizontal and vertical. In NOSQL systems, **horizontal scalability** is generally used, where the distributed system is expanded by adding more nodes for data storage and processing as the volume of data grows. Vertical scalability, on the other hand, refers to expanding the storage and computing power of existing nodes. In NOSQL systems, horizontal scalability is employed while the system is operational, so techniques for distributing the existing data among new nodes without interrupting system operation are necessary. We will discuss some of these techniques in Sections 24.3 through 24.6 when we discuss specific systems.
2. **Availability, Replication and Eventual Consistency:** Many applications that use NOSQL systems require continuous system availability. To accomplish this, data is replicated over two or more nodes in a transparent manner, so that if one node fails, the data is still available on other nodes. Replication improves data availability and can also improve read performance, because read requests can often be serviced from any of the replicated data nodes. However, write performance becomes more cumbersome because an update must be applied to every copy of the replicated data items; this can slow down write performance if serializable consistency is required (see Section 23.3). Many NOSQL applications do not require serializable



consistency, so more relaxed forms of consistency known as **eventual consistency** are used. We discuss this in more detail in Section 24.2.

3. **Replication Models:** Two major replication models are used in NOSQL systems: master-slave and master-master replication. **Master-slave replication** requires one copy to be the master copy; all write operations must be applied to the master copy and then propagated to the slave copies, usually using eventual consistency (the slave copies will *eventually* be the same as the master copy). For read, the master-slave paradigm can be configured in various ways. One configuration requires all reads to also be at the master copy, so this would be similar to the primary site or primary copy methods of distributed concurrency control (see Section 23.3.1), with similar advantages and disadvantages. Another configuration would allow reads at the slave copies but would not guarantee that the values are the latest writes, since writes to the slave nodes can be done after they are applied to the master copy. The **master-master replication** allows reads and writes at any of the replicas but may not guarantee that reads at nodes that store different copies see the same values. Different users may write the same data item concurrently at different nodes of the system, so the values of the item will be temporarily inconsistent. A reconciliation method to resolve conflicting write operations of the same data item at different nodes must be implemented as part of the master-master replication scheme.
4. **Sharding of Files:** In many NOSQL applications, files (or collections of data objects) can have many millions of records (or documents or objects), and these records can be accessed concurrently by thousands of users. So it is not practical to store the whole file in one node. **Sharding** (also known as **horizontal partitioning** ; see Section 23.2) of the file records is often employed in NOSQL systems. This serves to distribute the load of accessing the file records to multiple nodes. The combination of sharding the file records and replicating the shards works in tandem to improve load balancing as well as data availability. We will discuss some of the sharding techniques in Sections 24.3 through 24.6 when we discuss specific systems.
5. **High-Performance Data Access:** In many NOSQL applications, it is necessary to find individual records or objects (data items) from among the millions of data records or objects in a file. To achieve this, most systems use one of two techniques: hashing or range partitioning on object keys. The majority of accesses to an object will be by providing the key value rather than by using complex query conditions. The object key is similar to the concept of object id (see Section 12.1). In **hashing**, a hash function  $h(K)$  is applied to the key  $K$ , and the location of the object with key  $K$  is determined by the value of  $h(K)$ . In **range partitioning**, the location is determined via a range of key values; for example, location  $i$  would hold the objects whose key values  $K$  are in the range  $Ki_{\min} \leq K \leq Ki_{\max}$ . In applications that require range queries, where multiple objects within a range of key values are retrieved, range partitioned is preferred. Other indexes can also be used to locate objects based on attribute conditions different from the key  $K$ . We

will discuss some of the hashing, partitioning, and indexing techniques in Sections 24.3 through 24.6 when we discuss specific systems.

### **NOSQL characteristics related to data models and query languages.**

NOSQL systems emphasize performance and flexibility over modeling power and complex querying. We discuss some of these characteristics next.

1. **Not Requiring a Schema:** The flexibility of not requiring a schema is achieved in many NOSQL systems by allowing semi-structured, self-describing data (see Section 13.1). The users can specify a partial schema in some systems to improve storage efficiency, but it is *not required to have a schema* in most of the NOSQL systems. As there may not be a schema to specify constraints, any constraints on the data would have to be programmed in the application programs that access the data items. There are various languages for describing semistructured data, such as JSON (JavaScript Object Notation) and XML (Extensible Markup Language; see Chapter 13). JSON is used in several NOSQL systems, but other methods for describing semi-structured data can also be used. We will discuss JSON in Section 24.3 when we present document-based NOSQL systems.
2. **Less Powerful Query Languages:** Many applications that use NOSQL systems may not require a powerful query language such as SQL, because search (read) queries in these systems often locate single objects in a single file based on their object keys. NOSQL systems typically provide a set of functions and operations as a programming API (application programming interface), so reading and writing the data objects is accomplished by calling the appropriate operations by the programmer. In many cases, the operations are called **CRUD operations**, for Create, Read, Update, and Delete. In other cases, they are known as **SCRUD** because of an added Search (or Find) operation. Some NOSQL systems also provide a high-level query language, but it may not have the full power of SQL; only a subset of SQL querying capabilities would be provided. In particular, many NOSQL systems do not provide join operations as part of the query language itself; the joins need to be implemented in the application programs.
3. **Versioning:** Some NOSQL systems provide storage of multiple versions of the data items, with the timestamps of when the data version was created. We will discuss this aspect in Section 24.5 when we present column-based NOSQL systems.

In the next section, we give an overview of the various categories of NOSQL systems.

### **24.1.3 Categories of NOSQL Systems**

NOSQL systems have been characterized into four major categories, with some additional categories that encompass other types of systems. The most common categorization lists the following four major categories:

1. **Document-based NOSQL systems:** These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation). Documents are accessible via their document id, but can also be accessed rapidly using other indexes.
2. **NOSQL key-value stores:** These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.
3. **Column-based or wide column NOSQL systems:** These systems partition a table by column into column families (a form of vertical partitioning; see Section 23.2), where each column family is stored in its own files. They also allow versioning of data values.
4. **Graph-based NOSQL systems:** Data is represented as graphs, and related nodes can be found by traversing the edges using path expressions.

Additional categories can be added as follows to include some systems that are not easily categorized into the above four categories, as well as some other types of systems that have been available even before the term NOSQL became widely used.

5. **Hybrid NOSQL systems:** These systems have characteristics from two or more of the above four categories.
6. **Object databases:** These systems were discussed in Chapter 12.
7. **XML databases:** We discussed XML in Chapter 13.

Even keyword-based search engines store large amounts of data with fast search access, so the stored data can be considered as large NOSQL big data stores.

The rest of this chapter is organized as follows. In each of Sections 24.3 through 24.6, we will discuss one of the four main categories of NOSQL systems, and elaborate further on which characteristics each category focuses on. Before that, in Section 24.2, we discuss in more detail the concept of eventual consistency, and we discuss the associated CAP theorem.

## 24.2 The CAP Theorem

When we discussed concurrency control in distributed databases in Section 23.3, we assumed that the distributed database system (DDBS) is required to enforce the ACID properties (atomicity, consistency, isolation, durability) of transactions that are running concurrently (see Section 20.3). In a system with data replication, concurrency control becomes more complex because there can be multiple copies of each data item. So if an update is applied to one copy of an item, it must be applied to all other copies in a consistent manner. The possibility exists that one copy of an item  $X$  is updated by a transaction  $T_1$  whereas another copy is updated by a transaction  $T_2$ , so two inconsistent copies of the same item exist at two different nodes in the distributed system. If two other transactions  $T_3$  and  $T_4$  want to read  $X$ , each may read a different copy of item  $X$ .

We saw in Section 23.3 that there are distributed concurrency control methods that do not allow this inconsistency among copies of the same data item, thus enforcing serializability and hence the isolation property in the presence of replication. However, these techniques often come with high overhead, which would defeat the purpose of creating multiple copies to improve performance and availability in distributed database systems such as NOSQL. In the field of distributed systems, there are various levels of consistency among replicated data items, from weak consistency to strong consistency. Enforcing serializability is considered the strongest form of consistency, but it has high overhead so it can reduce performance of read and write operations and hence adversely affect system performance.

The CAP theorem, which was originally introduced as the CAP principle, can be used to explain some of the competing requirements in a distributed system with replication. The three letters in CAP refer to three desirable properties of distributed systems with replicated data: **consistency** (among replicated copies), **availability** (of the system for read and write operations) and **partition tolerance** (in the face of the nodes in the system being partitioned by a network fault). *Availability* means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. *Partition tolerance* means that the system can continue operating if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. *Consistency* means that the nodes will have the same copies of a replicated data item visible for various transactions.

It is important to note here that the use of the word *consistency* in CAP and its use in ACID *do not refer to the same identical concept*. In CAP, the term *consistency* refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In ACID, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema. However, if we consider that the consistency of replicated copies is a *specified constraint*, then the two uses of the term *consistency* would be related.

The **CAP theorem** states that it *is not possible to guarantee all three* of the desirable properties—consistency, availability, and partition tolerance—at the same time in a distributed system with data replication. If this is the case, then the distributed system designer would have to choose two properties out of the three to guarantee. It is generally assumed that in many traditional (SQL) applications, guaranteeing consistency through the ACID properties is important. On the other hand, in a NOSQL distributed data store, a weaker consistency level is often acceptable, and guaranteeing the other two properties (availability, partition tolerance) is important. Hence, weaker consistency levels are often used in NOSQL system instead of guaranteeing serializability. In particular, a form of consistency known as **eventual consistency** is often adopted in NOSQL systems. In Sections 24.3 through 24.6, we will discuss some of the consistency models used in specific NOSQL systems.

The next four sections of this chapter discuss the characteristics of the four main categories of NOSQL systems. We discuss document-based NOSQL systems in Section 24.3, and we use MongoDB as a representative system. In Section 24.4, we discuss

NOSQL systems known as key-value stores. In Section 24.5, we give an overview of column-based NOSQL systems, with a discussion of Hbase as a representative system. Finally, we introduce graph-based NOSQL systems in Section 24.6.

## 24.3 Document-Based NOSQL Systems and MongoDB

Document-based or document-oriented NOSQL systems typically store data as **collections** of similar **documents**. These types of systems are also sometimes known as **document stores**. The individual documents somewhat resemble *complex objects* (see Section 12.3) or XML documents (see Chapter 13), but a major difference between document-based systems versus object and object-relational systems and XML is that there is no requirement to specify a schema—rather, the documents are specified as **self-describing data** (see Section 13.1). Although the documents in a collection should be *similar*, they can have different data elements (attributes), and new documents can have new data elements that do not exist in any of the current documents in the collection. The system basically extracts the data element names from the self-describing documents in the collection, and the user can request that the system create indexes on some of the data elements. Documents can be specified in various formats, such as XML (see Chapter 13). A popular language to specify documents in NOSQL systems is **JSON** (JavaScript Object Notation).

There are many document-based NOSQL systems, including MongoDB and CouchDB, among many others. We will give an overview of MongoDB in this section. It is important to note that different systems can use different models, languages, and implementation methods, but giving a complete survey of all document-based NOSQL systems is beyond the scope of our presentation.

### 24.3.1 MongoDB Data Model

MongoDB documents are stored in BSON (Binary JSON) format, which is a variation of JSON with some additional data types and is more efficient for storage than JSON. Individual **documents** are stored in a **collection**. We will use a simple example based on our COMPANY database that we used throughout this book. The operation `createCollection` is used to create each collection. For example, the following command can be used to create a collection called **project** to hold PROJECT objects from the COMPANY database (see Figures 5.5 and 5.6):

```
db.createCollection("project", { capped : true, size : 1310720, max : 500 } )
```

The first parameter “project” is the **name** of the collection, which is followed by an optional document that specifies **collection options**. In our example, the collection is **capped**; this means it has upper limits on its storage space (**size**) and number of documents (**max**). The capping parameters help the system choose the storage options for each collection. There are other collection options, but we will not discuss them here.

For our example, we will create another document collection called **worker** to hold information about the EMPLOYEES who work on each project; for example:

```
db.createCollection("worker", { capped : true, size : 5242880, max : 2000 } ) )
```

Each document in a collection has a unique **ObjectId** field, called **\_id**, which is automatically indexed in the collection unless the user explicitly requests no index for the **\_id** field. The value of ObjectId can be *specified by the user*, or it can be *system-generated* if the user does not specify an **\_id** field for a particular document. *System-generated* ObjectIds have a specific format, which combines the timestamp when the object is created (4 bytes, in an internal MongoDB format), the node id (3 bytes), the process id (2 bytes), and a counter (3 bytes) into a 16-byte Id value. *User-generated* ObjectIds can have any value specified by the user as long as it uniquely identifies the document and so these Ids are similar to primary keys in relational systems.

A collection does not have a schema. The structure of the data fields in documents is chosen based on how documents will be accessed and used, and the user can choose a normalized design (similar to normalized relational tuples) or a denormalized design (similar to XML documents or complex objects). Interdocument references can be specified by storing in one document the ObjectId or ObjectIds of other related documents. Figure 24.1(a) shows a simplified MongoDB document showing some of the data from Figure 5.6 from the COMPANY database example that is used throughout the book. In our example, the **\_id** values are user-defined, and the documents whose **\_id** starts with P (for project) will be stored in the “project” collection, whereas those whose **\_id** starts with W (for worker) will be stored in the “worker” collection.

In Figure 24.1(a), the workers information is *embedded in the project document*; so there is no need for the “worker” collection. This is known as the *denormalized pattern*, which is similar to creating a complex object (see Chapter 12) or an XML document (see Chapter 13). A list of values that is enclosed in *square brackets* [ ... ] within a document represents a field whose value is an **array**.

Another option is to use the design in Figure 24.1(b), where *worker references* are embedded in the project document, but the worker documents themselves are stored in a separate “worker” collection. A third option in Figure 24.1(c) would use a normalized design, similar to First Normal Form relations (see Section 14.3.4). The choice of which design option to use depends on how the data will be accessed.

It is important to note that the simple design in Figure 24.1(c) *is not the general normalized design* for a many-to-many relationship, such as the one between employees and projects; rather, we would need three collections for “project”, “employee”, and “works\_on”, as we discussed in detail in Section 9.1. Many of the design tradeoffs that were discussed in Chapters 9 and 14 (for first normal form relations and for ER-to-relational mapping options), and Chapters 12 and 13 (for complex objects and XML) are applicable for choosing the appropriate design for document structures

**Figure 24.1**

Example of simple documents in MongoDB.

- (a) Denormalized document design with embedded subdocuments.  
 (b) Embedded array of document references.  
 (c) Normalized documents.

**(a) project document with an array of embedded workers:**

```
{
  _id:          "P1",
  Pname:        "ProductX",
  Plocation:    "Bellaire",
  Workers: [
    { Ename: "John Smith",
      Hours: 32.5
    },
    { Ename: "Joyce English",
      Hours: 20.0
    }
  ]
};
```

**(b) project document with an embedded array of worker ids:**

```
{
  _id:          "P1",
  Pname:        "ProductX",
  Plocation:    "Bellaire",
  WorkerIds:    [ "W1", "W2" ]
}

{ _id:          "W1",
  Ename:        "John Smith",
  Hours:        32.5
}

{ _id:          "W2",
  Ename:        "Joyce English",
  Hours:        20.0
}
```

**(c) normalized project and worker documents (not a fully normalized design for M:N relationships):**

```
{
  _id:          "P1",
  Pname:        "ProductX",
  Plocation:    "Bellaire"
}

{ _id:          "W1",
  Ename:        "John Smith",
  ProjectId:    "P1",
  Hours:        32.5
}
```



```
{  _id:          "W2",
   Ename:        "Joyce English",
   ProjectId:    "P1",
   Hours:        20.0
}
```

**Figure 24.1  
(continued)**

Example of simple documents in MongoDB. (d) Inserting the documents in Figure 24.1(c) into their collections.

**(d) inserting the documents in (c) into their collections “project” and “worker”:**

```
db.project.insert( { _id: "P1", Pname: "ProductX", Plocation: "Bellaire" } )
db.worker.insert( [ { _id: "W1", Ename: "John Smith", ProjectId: "P1", Hours: 32.5 },
                    { _id: "W2", Ename: "Joyce English", ProjectId: "P1",
                      Hours: 20.0 } ] )
```

and document collections, so we will not repeat the discussions here. In the design in Figure 24.1(c), an EMPLOYEE who works on several projects would be represented by *multiple worker documents* with different `_id` values; each document would represent the employee *as worker for a particular project*. This is similar to the design decisions for XML schema design (see Section 13.6). However, it is again important to note that the typical document-based system *does not have a schema*, so the design rules would have to be followed whenever individual documents are inserted into a collection.

## 24.3.2 MongoDB CRUD Operations

MongoDb has several **CRUD operations**, where CRUD stands for (create, read, update, delete). Documents can be *created* and inserted into their collections using the **insert** operation, whose format is:

```
db.<collection_name>.insert(<document(s)>)
```

The parameters of the insert operation can include either a single document or an array of documents, as shown in Figure 24.1(d). The *delete* operation is called **remove**, and the format is:

```
db.<collection_name>.remove(<condition>)
```

The documents to be removed from the collection are specified by a Boolean condition on some of the fields in the collection documents. There is also an **update** operation, which has a condition to select certain documents, and a `$set` clause to specify the update. It is also possible to use the update operation to replace an existing document with another one but keep the same ObjectId.

For *read* queries, the main command is called **find**, and the format is:

```
db.<collection_name>.find(<condition>)
```

General Boolean conditions can be specified as `<condition>`, and the documents in the collection that return **true** are selected for the query result. For a full discussion of the MongoDB CRUD operations, see the MongoDB online documentation in the chapter references.



### 24.3.3 MongoDB Distributed Systems Characteristics

Most MongoDB updates are atomic if they refer to a single document, but MongoDB also provides a pattern for specifying transactions on multiple documents. Since MongoDB is a distributed system, the **two-phase commit** method is used to ensure atomicity and consistency of multidocument transactions. We discussed the atomicity and consistency properties of transactions in Section 20.3, and the two-phase commit protocol in Section 22.6.

**Replication in MongoDB.** The concept of **replica set** is used in MongoDB to create multiple copies of the same data set on different nodes in the distributed system, and it uses a variation of the **master-slave** approach for replication. For example, suppose that we want to replicate a particular document collection C. A replica set will have one **primary copy** of the collection C stored in one node N1, and at least one **secondary copy** (replica) of C stored at another node N2. Additional copies can be stored in nodes N3, N4, etc., as needed, but the cost of storage and update (write) increases with the number of replicas. The total number of participants in a replica set must be at least three, so if only one secondary copy is needed, a participant in the replica set known as an **arbiter** must run on the third node N3. The arbiter does not hold a replica of the collection but participates in **elections** to choose a new primary if the node storing the current primary copy fails. If the total number of members in a replica set is  $n$  (one primary plus  $i$  secondaries, for a total of  $n = i + 1$ ), then  $n$  must be an odd number; if it is not, an *arbiter* is added to ensure the election process works correctly if the primary fails. We discussed elections in distributed systems in Section 23.3.1.

In MongoDB replication, all write operations must be applied to the primary copy and then propagated to the secondaries. For read operations, the user can choose the particular **read preference** for their application. The *default read preference* processes all reads at the primary copy, so all read and write operations are performed at the primary node. In this case, secondary copies are mainly to make sure that the system continues operation if the primary fails, and MongoDB can ensure that every read request gets the latest document value. To increase read performance, it is possible to set the read preference so that *read requests can be processed at any replica* (primary or secondary); however, a read at a secondary is not guaranteed to get the latest version of a document because there can be a delay in propagating writes from the primary to the secondaries.

**Sharding in MongoDB.** When a collection holds a very large number of documents or requires a large storage space, storing all the documents in one node can lead to performance problems, particularly if there are many user operations accessing the documents concurrently using various CRUD operations. **Sharding** of the documents in the collection—also known as *horizontal partitioning*—divides the documents into disjoint partitions known as **shards**. This allows the system to add more nodes as needed by a process known as **horizontal scaling** of the distributed system (see Section 23.1.4), and to store the shards of the collection on different nodes to achieve load balancing. Each node will process only those operations pertaining to the documents in the shard stored at that node. Also, each

shard will contain fewer documents than if the entire collection were stored at one node, thus further improving performance.

There are two ways to partition a collection into shards in MongoDB—**range partitioning** and **hash partitioning**. Both require that the user specify a particular document field to be used as the basis for partitioning the documents into shards. The *partitioning field*—known as the **shard key** in MongoDB—must have two characteristics: it must exist in *every document* in the collection, and it must have an *index*. The ObjectId can be used, but any other field possessing these two characteristics can also be used as the basis for sharding. The values of the shard key are divided into **chunks** either through range partitioning or hash partitioning, and the documents are partitioned based on the chunks of shard key values.

*Range partitioning* creates the chunks by specifying a range of key values; for example, if the shard key values ranged from one to ten million, it is possible to create ten ranges—1 to 1,000,000; 1,000,001 to 2,000,000; ... ; 9,000,001 to 10,000,000—and each chunk would contain the key values in one range. *Hash partitioning* applies a hash function  $h(K)$  to each shard key  $K$ , and the partitioning of keys into chunks is based on the hash values (we discussed hashing and its advantages and disadvantages in Section 16.8). In general, if **range queries** are commonly applied to a collection (for example, retrieving all documents whose shard key value is between 200 and 400), then range partitioning is preferred because each range query will typically be submitted to a single node that contains all the required documents in one shard. If most searches retrieve one document at a time, hash partitioning may be preferable because it randomizes the distribution of shard key values into chunks.

When sharding is used, MongoDB queries are submitted to a module called the **query router**, which keeps track of which nodes contain which shards based on the particular partitioning method used on the shard keys. The query (CRUD operation) will be routed to the nodes that contain the shards that hold the documents that the query is requesting. If the system cannot determine which shards hold the required documents, the query will be submitted to all the nodes that hold shards of the collection. Sharding and replication are used together; sharding focuses on improving performance via load balancing and horizontal scalability, whereas replication focuses on ensuring system availability when certain nodes fail in the distributed system.

There are many additional details about the distributed system architecture and components of MongoDB, but a full discussion is outside the scope of our presentation. MongoDB also provides many other services in areas such as system administration, indexing, security, and data aggregation, but we will not discuss these features here. Full documentation of MongoDB is available online (see the bibliographic notes).

## 24.4 NOSQL Key-Value Stores

**Key-value stores** focus on high performance, availability, and scalability by storing data in a distributed storage system. The data model used in key-value stores is relatively simple, and in many of these systems, there is no query language but rather a

set of operations that can be used by the application programmers. The **key** is a unique identifier associated with a data item and is used to locate this data item rapidly. The **value** is the data item itself, and it can have very different formats for different key-value storage systems. In some cases, the value is just a *string of bytes* or an *array of bytes*, and the application using the key-value store has to interpret the structure of the data value. In other cases, some standard formatted data is allowed; for example, structured data rows (tuples) similar to relational data, or semistructured data using JSON or some other self-describing data format. Different key-value stores can thus store unstructured, semistructured, or structured data items (see Section 13.1). The main characteristic of key-value stores is the fact that every value (data item) must be associated with a unique key, and that retrieving the value by supplying the key must be very fast.

There are many systems that fall under the key-value store label, so rather than provide a lot of details on one particular system, we will give a brief introductory overview for some of these systems and their characteristics.

### 24.4.1 DynamoDB Overview

The DynamoDB system is an Amazon product and is available as part of Amazon's AWS/SDK platforms (Amazon Web Services/Software Development Kit). It can be used as part of Amazon's cloud computing services, for the data storage component.

**DynamoDB data model.** The basic data model in DynamoDB uses the concepts of tables, items, and attributes. A **table** in DynamoDB *does not have* a **schema**; it holds a collection of *self-describing items*. Each **item** will consist of a number of (attribute, value) pairs, and attribute values can be single-valued or multivalued. So basically, a table will hold a collection of items, and each item is a self-describing record (or object). DynamoDB also allows the user to specify the items in JSON format, and the system will convert them to the internal storage format of DynamoDB.

When a table is created, it is required to specify a **table name** and a **primary key**; the primary key will be used to rapidly locate the items in the table. Thus, the primary key is the **key** and the item is the **value** for the DynamoDB key-value store. The primary key attribute must exist in every item in the table. The primary key can be one of the following two types:

- **A single attribute.** The DynamoDB system will use this attribute to build a hash index on the items in the table. This is called a *hash type primary key*. The items are not ordered in storage on the value of the hash attribute.
- **A pair of attributes.** This is called a *hash and range type primary key*. The primary key will be a pair of attributes (A, B): attribute A will be used for hashing, and because there will be multiple items with the same value of A, the B values will be used for ordering the records with the same A value. A table with this type of key can have additional secondary indexes defined on its attributes. For example, if we want to store multiple versions of some type of items in a table, we could use ItemID as hash and Date or Timestamp (when the version was created) as range in a hash and range type primary key.

**DynamoDB Distributed Characteristics.** Because DynamoDB is proprietary, in the next subsection we will discuss the mechanisms used for replication, sharding, and other distributed system concepts in an open source key-value system called Voldemort. Voldemort is based on many of the techniques proposed for DynamoDB.

### 24.4.2 Voldemort Key-Value Distributed Data Store

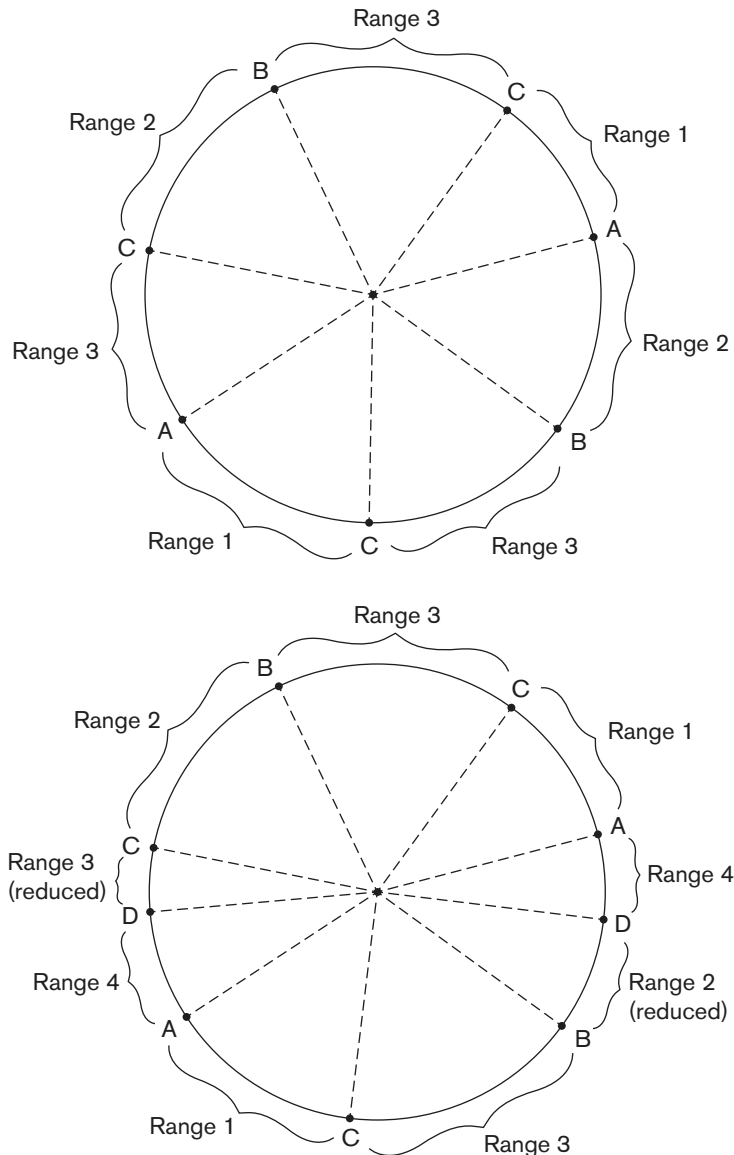
Voldemort is an open source system available through Apache 2.0 open source licensing rules. It is based on Amazon's DynamoDB. The focus is on high performance and horizontal scalability, as well as on providing replication for high availability and sharding for improving latency (response time) of read and write requests. All three of those features—replication, sharding, and horizontal scalability—are realized through a technique to distribute the key-value pairs among the nodes of a distributed cluster; this distribution is known as **consistent hashing**. Voldemort has been used by LinkedIn for data storage. Some of the features of Voldemort are as follows:

- **Simple basic operations.** A collection of (key, value) pairs is kept in a Voldemort **store**. In our discussion, we will assume the store is called *s*. The basic interface for data storage and retrieval is very simple and includes three operations: get, put, and delete. The operation *s.put(k, v)* inserts an item as a key-value pair with key *k* and value *v*. The operation *s.delete(k)* deletes the item whose key is *k* from the store, and the operation *v = s.get(k)* retrieves the value *v* associated with key *k*. The application can use these basic operations to build its own requirements. At the basic storage level, both keys and values are arrays of bytes (strings).
- **High-level formatted data values.** The values *v* in the (*k, v*) items can be specified in JSON (JavaScript Object Notation), and the system will convert between JSON and the internal storage format. Other data object formats can also be specified if the application provides the conversion (also known as **serialization**) between the user format and the storage format as a *Serializer class*. The *Serializer* class must be provided by the user and will include operations to convert the user format into a string of bytes for storage as a value, and to convert back a string (array of bytes) retrieved via *s.get(k)* into the user format. Voldemort has some built-in serializers for formats other than JSON.
- **Consistent hashing for distributing (key, value) pairs.** A variation of the data distribution algorithm known as **consistent hashing** is used in Voldemort for data distribution among the nodes in the distributed cluster of nodes. A hash function  $h(k)$  is applied to the key *k* of each (*k, v*) pair, and  $h(k)$  determines where the item will be stored. The method assumes that  $h(k)$  is an integer value, usually in the range 0 to  $Hmax = 2^{n-1}$ , where *n* is chosen based on the desired range for the hash values. This method is best visualized by considering the range of all possible integer hash values 0 to *Hmax* to be evenly distributed on a circle (or ring). The nodes in the distributed system are then also located on the same ring; usually each node will have several locations on the ring (see Figure 24.2). The positioning of the points on the ring that represent the nodes is done in a pseudorandom manner.

An item  $(k, v)$  will be stored on the node whose position in the ring *follows* the position of  $h(k)$  on the ring *in a clockwise direction*. In Figure 24.2(a), we assume there are three nodes in the distributed cluster labeled A, B, and C, where node C has a bigger capacity than nodes A and B. In a typical system, there will be many more nodes. On the circle, two instances each of A and B are placed, and three instances of C (because of its higher capacity), in a pseudorandom manner to cover the circle. Figure 24.2(a) indicates which  $(k, v)$  items are placed in which nodes based on the  $h(k)$  values.

**Figure 24.2**

Example of consistent hashing. (a) Ring having three nodes A, B, and C, with C having greater capacity. The  $h(k)$  values that map to the circle points in *range 1* have their  $(k, v)$  items stored in node A, *range 2* in node B, *range 3* in node C. (b) Adding a node D to the ring. Items in *range 4* are moved to the node D from node B (*range 2* is reduced) and node C (*range 3* is reduced).



- The  $h(k)$  values that fall in the parts of the circle marked as *range 1* in Figure 24.2(a) will have their  $(k, v)$  items stored in node A because that is the node whose label follows  $h(k)$  on the ring in a clockwise direction; those in *range 2* are stored in node B; and those in *range 3* are stored in node C. This scheme allows *horizontal scalability* because when a new node is added to the distributed system, it can be added in one or more locations on the ring depending on the node capacity. Only a limited percentage of the  $(k, v)$  items will be reassigned to the new node from the existing nodes based on the consistent hashing placement algorithm. Also, those items assigned to the new node may not all come from only one of the existing nodes because the new node can have multiple locations on the ring. For example, if a node D is added and it has two placements on the ring as shown in Figure 24.2(b), then some of the items from nodes B and C would be moved to node D. The items whose keys hash to *range 4* on the circle (see Figure 24.2(b)) would be migrated to node D. This scheme also allows *replication* by placing the number of specified replicas of an item on successive nodes on the ring in a clockwise direction. The *sharding* is built into the method, and different items in the store (file) are located on different nodes in the distributed cluster, which means the items are horizontally partitioned (sharded) among the nodes in the distributed system. When a node fails, its load of data items can be distributed to the other existing nodes whose labels follow the labels of the failed node in the ring. And nodes with higher capacity can have more locations on the ring, as illustrated by node C in Figure 24.2(a), and thus store more items than smaller-capacity nodes.
- **Consistency and versioning.** Voldemort uses a method similar to the one developed for DynamoDB for consistency in the presence of replicas. Basically, concurrent write operations are allowed by different processes so there could exist two or more different values associated with the same key at different nodes when items are replicated. Consistency is achieved when the item is read by using a technique known as *versioning and read repair*. Concurrent writes are allowed, but each write is associated with a *vector clock* value. When a read occurs, it is possible that different versions of the same value (associated with the same key) are read from different nodes. If the system can reconcile to a single final value, it will pass that value to the read; otherwise, more than one version can be passed back to the application, which will reconcile the various versions into one version based on the application semantics and give this reconciled value back to the nodes.

### 24.4.3 Examples of Other Key-Value Stores

In this section, we briefly review three other key-value stores. It is important to note that there are many systems that can be classified in this category, and we can only mention a few of these systems.

**Oracle key-value store.** Oracle has one of the well-known SQL relational database systems, and Oracle also offers a system based on the key-value store concept; this system is called the **Oracle NoSQL Database**.



**Redis key-value cache and store.** Redis differs from the other systems discussed here because it caches its data in main memory to further improve performance. It offers master-slave replication and high availability, and it also offers persistence by backing up the cache to disk.

**Apache Cassandra.** Cassandra is a NOSQL system that is not easily categorized into one category; it is sometimes listed in the column-based NOSQL category (see Section 24.5) or in the key-value category. It offers features from several NOSQL categories and is used by Facebook as well as many other customers.

## 24.5 Column-Based or Wide Column NOSQL Systems

Another category of NOSQL systems is known as **column-based** or **wide column** systems. The Google distributed storage system for big data, known as **BigTable**, is a well-known example of this class of NOSQL systems, and it is used in many Google applications that require large amounts of data storage, such as Gmail. BigTable uses the **Google File System (GFS)** for data storage and distribution. An open source system known as **Apache Hbase** is somewhat similar to Google BigTable, but it typically uses **HDFS (Hadoop Distributed File System)** for data storage. HDFS is used in many cloud computing applications, as we shall discuss in Chapter 25. Hbase can also use Amazon's **Simple Storage System** (known as **S3**) for data storage. Another well-known example of column-based NOSQL systems is Cassandra, which we discussed briefly in Section 24.4.3 because it can also be characterized as a key-value store. We will focus on Hbase in this section as an example of this category of NOSQL systems.

BigTable (and Hbase) is sometimes described as a *sparse multidimensional distributed persistent sorted map*, where the word *map* means a *collection of (key, value) pairs* (the key is *mapped* to the value). One of the main differences that distinguish column-based systems from key-value stores (see Section 24.4) is the *nature of the key*. In column-based systems such as Hbase, the key is *multidimensional* and so has several components: typically, a combination of table name, row key, column, and timestamp. As we shall see, the column is typically composed of two components: column family and column qualifier. We discuss these concepts in more detail next as they are realized in Apache Hbase.

### 24.5.1 Hbase Data Model and Versioning

**Hbase data model.** The data model in Hbase organizes data using the concepts of *namespaces*, *tables*, *column families*, *column qualifiers*, *columns*, *rows*, and *data cells*. A column is identified by a combination of (column family:column qualifier). Data is stored in a self-describing form by associating columns with data values, where data values are strings. Hbase also stores *multiple versions* of a data item, with a *timestamp* associated with each version, so versions and timestamps are also

part of the Hbase data model (this is similar to the concept of attribute versioning in temporal databases, which we shall discuss in Section 26.2). As with other NOSQL systems, unique keys are associated with stored data items for fast access, but the keys identify *cells* in the storage system. Because the focus is on high performance when storing huge amounts of data, the data model includes some storage-related concepts. We discuss the Hbase data modeling concepts and define the terminology next. It is important to note that the use of the words *table*, *row*, and *column* is not identical to their use in relational databases, but the uses are related.

- **Tables and Rows.** Data in Hbase is stored in **tables**, and each table has a table name. Data in a table is stored as self-describing **rows**. Each row has a unique **row key**, and row keys are strings that must have the property that they can be lexicographically ordered, so characters that do not have a lexicographic order in the character set cannot be used as part of a row key.
- **Column Families, Column Qualifiers, and Columns.** A table is associated with one or more **column families**. Each column family will have a name, and the column families associated with a table *must be specified* when the table is created and cannot be changed later. Figure 24.3(a) shows how a table may be created; the table name is followed by the names of the column families associated with the table. When the data is loaded into a table, each column family can be associated with many **column qualifiers**, but the column qualifiers *are not specified* as part of creating a table. So the column qualifiers make the model a self-describing data model because the qualifiers can be dynamically specified as new rows are created and inserted into the table. A **column** is specified by a combination of ColumnFamily:ColumnQualifier. Basically, column families are a way of grouping together related columns (attributes in relational terminology) for storage purposes, except that the column qualifier names are not specified during table creation. Rather, they are specified when the data is created and stored in rows, so the data is *self-describing* since any column qualifier name can be used in a new row of data (see Figure 24.3(b)). However, it is important that the application programmers know which column qualifiers belong to each column family, even though they have the flexibility to create new column qualifiers on the fly when new data rows are created. The concept of column family is somewhat similar to *vertical partitioning* (see Section 23.2), because columns (attributes) that are accessed together because they belong to the same column family are stored in the same files. Each column family of a table is stored in its own files using the HDFS file system.
- **Versions and Timestamps.** Hbase can keep several **versions** of a data item, along with the **timestamp** associated with each version. The timestamp is a long integer number that represents the system time when the version was created, so newer versions have larger timestamp values. Hbase uses midnight ‘January 1, 1970 UTC’ as timestamp value zero, and uses a long integer that measures the number of milliseconds since that time as the system timestamp value (this is similar to the value returned by the Java utility `java.util.Date.getTime()` and is also used in MongoDB). It is also possible for



**Figure 24.3**

Examples in Hbase. (a) Creating a table called EMPLOYEE with three column families: Name, Address, and Details. (b) Inserting some in the EMPLOYEE table; different rows can have different self-describing column qualifiers (Fname, Lname, Nickname, Mname, Minit, Suffix, ... for column family Name; Job, Review, Supervisor, Salary for column family Details). (c) Some CRUD operations of Hbase.

**(a) creating a table:**

```
create 'EMPLOYEE', 'Name', 'Address', 'Details'
```

**(b) inserting some row data in the EMPLOYEE table:**

```
put 'EMPLOYEE', 'row1', 'Name:Fname', 'John'
put 'EMPLOYEE', 'row1', 'Name:Lname', 'Smith'
put 'EMPLOYEE', 'row1', 'Name:Nickname', 'Johnny'
put 'EMPLOYEE', 'row1', 'Details:Job', 'Engineer'
put 'EMPLOYEE', 'row1', 'Details:Review', 'Good'
put 'EMPLOYEE', 'row2', 'Name:Fname', 'Alicia'
put 'EMPLOYEE', 'row2', 'Name:Lname', 'Zelaya'
put 'EMPLOYEE', 'row2', 'Name:MName', 'Jennifer'
put 'EMPLOYEE', 'row2', 'Details:Job', 'DBA'
put 'EMPLOYEE', 'row2', 'Details:Supervisor', 'James Borg'
put 'EMPLOYEE', 'row3', 'Name:Fname', 'James'
put 'EMPLOYEE', 'row3', 'Name:Minit', 'E'
put 'EMPLOYEE', 'row3', 'Name:Lname', 'Borg'
put 'EMPLOYEE', 'row3', 'Name:Suffix', 'Jr.'
put 'EMPLOYEE', 'row3', 'Details:Job', 'CEO'
put 'EMPLOYEE', 'row3', 'Details:Salary', '1,000,000'
```

**(c) Some Hbase basic CRUD operations:**

Creating a table: `create <tablename>, <column family>, <column family>, ...`

Inserting Data: `put <tablename>, <rowid>, <column family>:<column qualifier>, <value>`

Reading Data (all data in a table): `scan <tablename>`

Retrieve Data (one item): `get <tablename>,<rowid>`

the user to define the timestamp value explicitly in a Date format rather than using the system-generated timestamp.

- **Cells.** A **cell** holds a basic data item in Hbase. The key (address) of a cell is specified by a combination of (table, rowid, columnfamily, columnqualifier, timestamp). If timestamp is left out, the latest version of the item is retrieved unless a default number of versions is specified, say the latest three versions. The default number of versions to be retrieved, as well as the default number of versions that the system needs to keep, are parameters that can be specified during table creation.
- **Namespaces.** A **namespace** is a collection of tables. A namespace basically specifies a collection of one or more tables that are typically used together by user applications, and it corresponds to a database that contains a collection of tables in relational terminology.

### 24.5.2 Hbase CRUD Operations

Hbase has low-level CRUD (create, read, update, delete) operations, as in many of the NOSQL systems. The formats of some of the basic CRUD operations in Hbase are shown in Figure 24.3(c).

Hbase only provides low-level CRUD operations. It is the responsibility of the application programs to implement more complex operations, such as joins between rows in different tables. The *create* operation creates a new table and specifies one or more column families associated with that table, but it does not specify the column qualifiers, as we discussed earlier. The *put* operation is used for inserting new data or new versions of existing data items. The *get* operation is for retrieving the data associated with a single row in a table, and the *scan* operation retrieves all the rows.

### 24.5.3 Hbase Storage and Distributed System Concepts

Each Hbase table is divided into a number of **regions**, where each region will hold a *range* of the row keys in the table; this is why the row keys must be lexicographically ordered. Each region will have a number of **stores**, where each column family is assigned to one store within the region. Regions are assigned to **region servers** (storage nodes) for storage. A **master server** (master node) is responsible for monitoring the region servers and for splitting a table into regions and assigning regions to region servers.

Hbase uses the **Apache Zookeeper** open source system for services related to managing the naming, distribution, and synchronization of the Hbase data on the distributed Hbase server nodes, as well as for coordination and replication services. Hbase also uses Apache HDFS (Hadoop Distributed File System) for distributed file services. So Hbase is built on top of both HDFS and Zookeeper. Zookeeper can itself have several replicas on several nodes for availability, and it keeps the data it needs in main memory to speed access to the master servers and region servers.

We will not cover the many additional details about the distributed system architecture and components of Hbase; a full discussion is outside the scope of our presentation. Full documentation of Hbase is available online (see the bibliographic notes).

## 24.6 NOSQL Graph Databases and Neo4j

Another category of NOSQL systems is known as **graph databases** or **graph-oriented NOSQL** systems. The data is represented as a graph, which is a collection of vertices (nodes) and edges. Both nodes and edges can be labeled to indicate the types of entities and relationships they represent, and it is generally possible to store data associated with both individual nodes and individual edges. Many systems can be categorized as graph databases. We will focus our discussion on one particular system, Neo4j, which is used in many applications. Neo4j is an open source system, and it is implemented in Java. We will discuss the Neo4j data model

in Section 24.6.1, and give an introduction to the Neo4j querying capabilities in Section 24.6.2. Section 24.6.3 gives an overview of the distributed systems and some other characteristics of Neo4j.

### 24.6.1 Neo4j Data Model

The data model in Neo4j organizes data using the concepts of **nodes** and **relationships**. Both nodes and relationships can have **properties**, which store the data items associated with nodes and relationships. Nodes can have **labels**; the nodes that have the *same label* are grouped into a collection that identifies a subset of the nodes in the database graph for querying purposes. A node can have zero, one, or several labels. Relationships are directed; each relationship has a *start node* and *end node* as well as a **relationship type**, which serves a similar role to a node label by identifying similar relationships that have the same relationship type. Properties can be specified via a **map pattern**, which is made of one or more “name : value” pairs enclosed in curly brackets; for example {Lname : ‘Smith’, Fname : ‘John’, Minit : ‘B’}.

In conventional graph theory, nodes and relationships are generally called *vertices* and *edges*. The Neo4j graph data model somewhat resembles how data is represented in the ER and EER models (see Chapters 3 and 4), but with some notable differences. Comparing the Neo4j graph model with ER/EER concepts, nodes correspond to *entities*, node labels correspond to *entity types and subclasses*, relationships correspond to *relationship instances*, relationship types correspond to *relationship types*, and properties correspond to *attributes*. One notable difference is that a relationship is *directed* in Neo4j, but is not in ER/EER. Another is that a node may have no label in Neo4j, which is not allowed in ER/EER because every entity must belong to an entity type. A third crucial difference is that the graph model of Neo4j is used as a basis for an actual high-performance distributed database system whereas the ER/EER model is mainly used for database design.

Figure 24.4(a) shows how a few nodes can be created in Neo4j. There are various ways in which nodes and relationships can be created; for example, by calling appropriate Neo4j operations from various Neo4j APIs. We will just show the high-level syntax for creating nodes and relationships; to do so, we will use the Neo4j CREATE command, which is part of the high-level declarative query language **Cypher**. Neo4j has many options and variations for creating nodes and relationships using various scripting interfaces, but a full discussion is outside the scope of our presentation.

- **Labels and properties.** When a node is created, the node label can be specified. It is also possible to create nodes without any labels. In Figure 24.4(a), the node labels are EMPLOYEE, DEPARTMENT, PROJECT, and LOCATION, and the created nodes correspond to some of the data from the COMPANY database in Figure 5.6 with a few modifications; for example, we use EmpId instead of SSN, and we only include a small subset of the data for illustration purposes. Properties are enclosed in curly brackets { ... }. It is possible that some nodes have multiple labels; for example the same node can be labeled as PERSON and EMPLOYEE and MANAGER by listing all the labels separated by the colon symbol as follows: PERSON:EMPLOYEE:MANAGER. Having multiple labels is similar to an entity belonging to an entity type (PERSON)

plus some subclasses of PERSON (namely EMPLOYEE and MANAGER) in the EER model (see Chapter 4) but can also be used for other purposes.

- **Relationships and relationship types.** Figure 24.4(b) shows a few example relationships in Neo4j based on the COMPANY database in Figure 5.6. The  $\rightarrow$  specifies the direction of the relationship, but the relationship can be traversed in either direction. The relationship types (labels) in Figure 24.4(b) are WorksFor, Manager, LocatedIn, and WorksOn; only relationships with the relationship type WorksOn have properties (Hours) in Figure 24.4(b).
- **Paths.** A **path** specifies a traversal of part of the graph. It is typically used as part of a query to specify a pattern, where the query will retrieve from the graph data that matches the pattern. A path is typically specified by a start node, followed by one or more relationships, leading to one or more end nodes that satisfy the pattern. It is somewhat similar to the concepts of path expressions that we discussed in Chapters 12 and 13 in the context of query languages for object databases (OQL) and XML (XPath and XQuery).
- **Optional Schema.** A **schema** is optional in Neo4j. Graphs can be created and used without a schema, but in Neo4j version 2.0, a few schema-related functions were added. The main features related to schema creation involve creating indexes and constraints based on the labels and properties. For example, it is possible to create the equivalent of a key constraint on a property of a label, so all nodes in the collection of nodes associated with the label must have unique values for that property.
- **Indexing and node identifiers.** When a node is created, the Neo4j system creates an internal unique system-defined identifier for each node. To retrieve individual nodes using other properties of the nodes efficiently, the user can create **indexes** for the collection of nodes that have a particular label. Typically, one or more of the properties of the nodes in that collection can be indexed. For example, Empid can be used to index nodes with the EMPLOYEE label, Dno to index the nodes with the DEPARTMENT label, and Pno to index the nodes with the PROJECT label.

## 24.6.2 The Cypher Query Language of Neo4j

Neo4j has a high-level query language, Cypher. There are declarative commands for creating nodes and relationships (see Figures 24.4(a) and (b)), as well as for finding nodes and relationships based on specifying patterns. Deletion and modification of data is also possible in Cypher. We introduced the CREATE command in the previous section, so we will now give a brief overview of some of the other features of Cypher.

A Cypher query is made up of *clauses*. When a query has several clauses, the result from one clause can be the input to the next clause in the query. We will give a flavor of the language by discussing some of the clauses using examples. Our presentation is not meant to be a detailed presentation on Cypher, just an introduction to some of the languages features. Figure 24.4(c) summarizes some of the main clauses that can be part of a Cyber query. The Cyber language can specify complex queries and updates on a graph database. We will give a few of examples to illustrate simple Cyber queries in Figure 24.4(d).

**Figure 24.4**

Examples in Neo4j using the Cypher language. (a) Creating some nodes. (b) Creating some relationships.

**(a) creating some nodes for the COMPANY data (from Figure 5.6):**

```
CREATE (e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})
CREATE (e2: EMPLOYEE, {Empid: '2', Lname: 'Wong', Fname: 'Franklin'})
CREATE (e3: EMPLOYEE, {Empid: '3', Lname: 'Zelaya', Fname: 'Alicia'})
CREATE (e4: EMPLOYEE, {Empid: '4', Lname: 'Wallace', Fname: 'Jennifer', Minit: 'S'})

...

CREATE (d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})
CREATE (d2: DEPARTMENT, {Dno: '4', Dname: 'Administration'})

...

CREATE (p1: PROJECT, {Pno: '1', Pname: 'ProductX'})
CREATE (p2: PROJECT, {Pno: '2', Pname: 'ProductY'})
CREATE (p3: PROJECT, {Pno: '10', Pname: 'Computerization'})
CREATE (p4: PROJECT, {Pno: '20', Pname: 'Reorganization'})

...

CREATE (loc1: LOCATION, {Lname: 'Houston'})
CREATE (loc2: LOCATION, {Lname: 'Stafford'})
CREATE (loc3: LOCATION, {Lname: 'Bellaire'})
CREATE (loc4: LOCATION, {Lname: 'Sugarland'})

...
```

**(b) creating some relationships for the COMPANY data (from Figure 5.6):**

```
CREATE (e1) - [ : WorksFor ] -> (d1)
CREATE (e3) - [ : WorksFor ] -> (d2)

...

CREATE (d1) - [ : Manager ] -> (e2)
CREATE (d2) - [ : Manager ] -> (e4)

...

CREATE (d1) - [ : LocatedIn ] -> (loc1)
CREATE (d1) - [ : LocatedIn ] -> (loc3)
CREATE (d1) - [ : LocatedIn ] -> (loc4)
CREATE (d2) - [ : LocatedIn ] -> (loc2)

...

CREATE (e1) - [ : WorksOn, {Hours: '32.5'} ] -> (p1)
CREATE (e1) - [ : WorksOn, {Hours: '7.5'} ] -> (p2)
CREATE (e2) - [ : WorksOn, {Hours: '10.0'} ] -> (p1)
CREATE (e2) - [ : WorksOn, {Hours: 10.0} ] -> (p2)
CREATE (e2) - [ : WorksOn, {Hours: '10.0'} ] -> (p3)
CREATE (e2) - [ : WorksOn, {Hours: 10.0} ] -> (p4)

...
```

**Figure 24.4 (continued)**

Examples in Neo4j using the Cypher language. (c) Basic syntax of Cypher queries. (d) Examples of Cypher queries.

**(c) Basic simplified syntax of some common Cypher clauses:**

Finding nodes and relationships that match a pattern: `MATCH <pattern>`  
 Specifying aggregates and other query variables: `WITH <specifications>`  
 Specifying conditions on the data to be retrieved: `WHERE <condition>`  
 Specifying the data to be returned: `RETURN <data>`  
 Ordering the data to be returned: `ORDER BY <data>`  
 Limiting the number of returned data items: `LIMIT <max number>`  
 Creating nodes: `CREATE <node, optional labels and properties>`  
 Creating relationships: `CREATE <relationship, relationship type and optional properties>`  
 Deletion: `DELETE <nodes or relationships>`  
 Specifying property values and labels: `SET <property values and labels>`  
 Removing property values and labels: `REMOVE <property values and labels>`

**(d) Examples of simple Cypher queries:**

1. `MATCH (d : DEPARTMENT {Dno: '5'}) - [ : LocatedIn ] -> (loc)`  
`RETURN d.Dname , loc.Lname`
2. `MATCH (e: EMPLOYEE {Empid: '2'}) - [ w: WorksOn ] -> (p)`  
`RETURN e.Ename , w.Hours, p.Pname`
3. `MATCH (e) - [ w: WorksOn ] -> (p: PROJECT {Pno: 2})`  
`RETURN p.Pname, e.Ename , w.Hours`
4. `MATCH (e) - [ w: WorksOn ] -> (p)`  
`RETURN e.Ename , w.Hours, p.Pname`  
`ORDER BY e.Ename`
5. `MATCH (e) - [ w: WorksOn ] -> (p)`  
`RETURN e.Ename , w.Hours, p.Pname`  
`ORDER BY e.Ename`  
`LIMIT 10`
6. `MATCH (e) - [ w: WorksOn ] -> (p)`  
`WITH e, COUNT(p) AS numOfprojs`  
`WHERE numOfprojs > 2`  
`RETURN e.Ename , numOfprojs`  
`ORDER BY numOfprojs`
7. `MATCH (e) - [ w: WorksOn ] -> (p)`  
`RETURN e , w, p`  
`ORDER BY e.Ename`  
`LIMIT 10`
8. `MATCH (e: EMPLOYEE {Empid: '2'})`  
`SET e.Job = 'Engineer'`

---

Query 1 in Figure 24.4(d) shows how to use the `MATCH` and `RETURN` clauses in a query, and the query retrieves the locations for department number 5. Match specifies the *pattern* and the *query variables* (*d* and *loc*) and `RETURN` specifies the query result to be retrieved by referring to the query variables. Query 2 has three variables (*e*, *w*, and *p*), and returns the projects and hours per week that the employee with

Empid = 2 works on. Query 3, on the other hand, returns the employees and hours per week who work on the project with Pno = 2. Query 4 illustrates the ORDER BY clause and returns all employees and the projects they work on, sorted by Ename. It is also possible to limit the number of returned results by using the LIMIT clause as in query 5, which only returns the first 10 answers.

Query 6 illustrates the use of WITH and aggregation, although the WITH clause can be used to separate clauses in a query even if there is no aggregation. Query 6 also illustrates the WHERE clause to specify additional conditions, and the query returns the employees who work on more than two projects, as well as the number of projects each employee works on. It is also common to return the nodes and relationships themselves in the query result, rather than the property values of the nodes as in the previous queries. Query 7 is similar to query 5 but returns the nodes and relationships only, and so the query result can be displayed as a graph using Neo4j's visualization tool. It is also possible to add or remove labels and properties from nodes. Query 8 shows how to add more properties to a node by adding a Job property to an employee node.

The above gives a brief flavor for the Cypher query language of Neo4j. The full language manual is available online (see the bibliographic notes).

### 24.6.3 Neo4j Interfaces and Distributed System Characteristics

Neo4j has other interfaces that can be used to create, retrieve, and update nodes and relationships in a graph database. It also has two main versions: the enterprise edition, which comes with additional capabilities, and the community edition. We discuss some of the additional features of Neo4j in this subsection.

- **Enterprise edition vs. community edition.** Both editions support the Neo4j graph data model and storage system, as well as the Cypher graph query language, and several other interfaces, including a high-performance native API, language drivers for several popular programming languages, such as Java, Python, PHP, and the REST (Representational State Transfer) API. In addition, both editions support ACID properties. The enterprise edition supports additional features for enhancing performance, such as caching and clustering of data and locking.
- **Graph visualization interface.** Neo4j has a graph visualization interface, so that a subset of the nodes and edges in a database graph can be displayed as a graph. This tool can be used to visualize query results in a graph representation.
- **Master-slave replication.** Neo4j can be configured on a cluster of distributed system nodes (computers), where one node is designated the master node. The data and indexes are fully replicated on each node in the cluster. Various ways of synchronizing the data between master and slave nodes can be configured in the distributed cluster.
- **Caching.** A main memory cache can be configured to store the graph data for improved performance.
- **Logical logs.** Logs can be maintained to recover from failures.



A full discussion of all the features and interfaces of Neo4j is outside the scope of our presentation. Full documentation of Neo4j is available online (see the bibliographic notes).

## 24.7 Summary

In this chapter, we discussed the class of database systems known as NOSQL systems, which focus on efficient storage and retrieval of large amounts of “big data.” Applications that use these types of systems include social media, Web links, user profiles, marketing and sales, posts and tweets, road maps and spatial data, and e-mail. The term *NOSQL* is generally interpreted as Not Only SQL—rather than NO to SQL—and is meant to convey that many applications need systems other than traditional relational SQL systems to augment their data management needs. These systems are distributed databases or distributed storage systems, with a focus on semistructured data storage, high performance, availability, data replication, and scalability rather than an emphasis on immediate data consistency, powerful query languages, and structured data storage.

In Section 24.1, we started with an introduction to NOSQL systems, their characteristics, and how they differ from SQL systems. Four general categories of NOSQL systems are document-based, key-value stores, column-based, and graph-based. In Section 24.2, we discussed how NOSQL systems approach the issue of consistency among multiple replicas (copies) by using the paradigm known as eventual consistency. We discussed the CAP theorem, which can be used to understand the emphasis of NOSQL systems on availability. In Sections 24.3 through 24.6, we presented an overview of each of the four main categories of NOSQL systems—starting with document-based systems in Section 24.3, followed by key-value stores in Section 24.4, then column-based systems in Section 24.5, and finally graph-based systems in Section 24.6. We also noted that some NOSQL systems may not fall neatly into a single category but rather use techniques that span two or more categories.

## Review Questions

- 24.1.** For which types of applications were NOSQL systems developed?
- 24.2.** What are the main categories of NOSQL systems? List a few of the NOSQL systems in each category.
- 24.3.** What are the main characteristics of NOSQL systems in the areas related to data models and query languages?
- 24.4.** What are the main characteristics of NOSQL systems in the areas related to distributed systems and distributed databases?
- 24.5.** What is the CAP theorem? Which of the three properties (consistency, availability, partition tolerance) are most important in NOSQL systems?



- 24.6.** What are the similarities and differences between using consistency in CAP versus using consistency in ACID?
- 24.7.** What are the data modeling concepts used in MongoDB? What are the main CRUD operations of MongoDB?
- 24.8.** Discuss how replication and sharding are done in MongoDB.
- 24.9.** Discuss the data modeling concepts in DynamoDB.
- 24.10.** Describe the consistent hashing schema for data distribution, replication, and sharding. How are consistency and versioning handled in Voldemort?
- 24.11.** What are the data modeling concepts used in column-based NOSQL systems and Hbase?
- 24.12.** What are the main CRUD operations in Hbase?
- 24.13.** Discuss the storage and distributed system methods used in Hbase.
- 24.14.** What are the data modeling concepts used in the graph-oriented NOSQL system Neo4j?
- 24.15.** What is the query language for Neo4j?
- 24.16.** Discuss the interfaces and distributed systems characteristics of Neo4j.

## Selected Bibliography

The original paper that described the Google BigTable distributed storage system is Chang et al. (2006), and the original paper that described the Amazon Dynamo key-value store system is DeCandia et al. (2007). There are numerous papers that compare various NOSQL systems with SQL (relational systems); for example, Parker et al. (2013). Other papers compare NOSQL systems to other NOSQL systems; for example Cattell (2010), Hecht and Jablonski (2011), and Abramova and Bernardino (2013).

The documentation, user manuals, and tutorials for many NOSQL systems can be found on the Web. Here are a few examples:

MongoDB tutorials: [docs.mongodb.org/manual/tutorial/](http://docs.mongodb.org/manual/tutorial/)  
 MongoDB manual: [docs.mongodb.org/manual/](http://docs.mongodb.org/manual/)  
 Voldemort documentation: [docs.project-voldemort.com/voldemort/](http://docs.project-voldemort.com/voldemort/)  
 Cassandra Web site: [cassandra.apache.org](http://cassandra.apache.org)  
 Hbase Web site: [hbase.apache.org](http://hbase.apache.org)  
 Neo4j documentation: [neo4j.com/docs/](http://neo4j.com/docs/)

In addition, numerous Web sites categorize NOSQL systems into additional sub-categories based on purpose; [nosql-database.org](http://nosql-database.org) is one example of such a site.

## Big Data Technologies Based on MapReduce and Hadoop<sup>1</sup>

The amount of data worldwide has been growing ever since the advent of the World Wide Web around 1994. The early search engines—namely, AltaVista (which was acquired by Yahoo in 2003 and which later became the Yahoo! search engine) and Lycos (which was also a search engine and a Web portal—were established soon after the Web came along. They were later overshadowed by the likes of Google and Bing. Then came an array of social networks such as Facebook, launched in 2004, and Twitter, founded in 2006. LinkedIn, a professional network launched in 2003, boasts over 250 million users worldwide. Facebook has over 1.3 billion users worldwide today; of these, about 800 million are active on Facebook daily. Twitter had an estimated 980 million users in early 2014 and it was reported to have reached the rate of 1 billion tweets per day in October 2012. These statistics are updated continually and are easily available on the Web.

One major implication of the establishment and exponential growth of the Web, which brought computing to laypeople worldwide, is that ordinary people started creating all types of transactions and content that generate new data. These users and consumers of multimedia data require systems to deliver user-specific data instantaneously from mammoth stores of data at the same time that they create huge amounts of data themselves. The result is an explosive growth in the amount of data generated and communicated over networks worldwide; in addition, businesses and governmental institutions electronically record every transaction of each customer, vendor, and supplier and thus have been accumulating data in so-called data warehouses (to be discussed in Chapter 29). Added to this mountain of data is the data

---

<sup>1</sup>We acknowledge the significant contribution of Harish Butani, member of the Hive Program Management Committee, and Balaji Palanisamy, University of Pittsburgh, to this chapter.

generated by sensors embedded in devices such as smartphones, energy smart meters, automobiles, and all kinds of gadgets and machinery that sense, create, and communicate data in the internet of things. And, of course, we must consider the data generated daily from satellite imagery and communication networks.

This phenomenal growth of data generation means that the amount of data in a single repository can be numbered in petabytes (10<sup>15</sup> bytes, which approximates to 2<sup>50</sup> bytes) or terabytes (e.g., 1,000 terabytes). The term *big data* has entered our common parlance and refers to such massive amounts of data. The McKinsey report<sup>2</sup> defines the term *big data* as datasets whose size exceeds the typical reach of a DBMS to capture, store, manage, and analyze that data. The meaning and implications of this data onslaught are reflected in some of the facts mentioned in the McKinsey report:

- A \$600 disk can store all of the world's music today.
- Every month, 30 billion of items of content are stored on Facebook.
- More data is stored in 15 of the 17 sectors of the U.S. economy than is stored in the Library of Congress, which, as of 2011, stored 235 terabytes of data.
- There is currently a need for over 140,000 deep-data-analysis positions and over 1.5 million data-savvy managers in the United States. Deep data analysis involves more knowledge discovery type analyses.

Big data is everywhere, so every sector of the economy stands to benefit by harnessing it appropriately with technologies that will help data users and managers make better decisions based on historical evidence. According to the McKinsey report,

If the U.S. healthcare [system] could use the big data creatively and effectively to drive efficiency and quality, we estimate that the potential value from data in the sector could be more than \$300 billion in value every year.

Big data has created countless opportunities to give consumers information in a timely manner—information that will prove useful in making decisions, discovering needs and improving performance, customizing products and services, giving decision makers more effective algorithmic tools, and creating value by innovations in terms of new products, services, and business models. IBM has corroborated this statement in a recent book,<sup>3</sup> which outlines why IBM has embarked on a worldwide mission of enterprise-wide big data analytics. The IBM book describes various types of analytics applications:

- **Descriptive and predictive analytics:** Descriptive analytics relates to reporting what has happened, analyzing the data that contributed to it to figure out why it happened, and monitoring new data to find out what is happening now. Predictive analytics uses statistical and data mining techniques (see Chapter 28) to make predictions about what will happen in the future.

---

<sup>2</sup>The introduction is largely based on the McKinsey (2012) report on big data from the McKinsey Global Institute.

<sup>3</sup>See IBM (2014): *Analytics Across the Enterprise: How IBM Realizes Business Value from Big Data and Analytics*.

- **Prescriptive analytics:** Refers to analytics that recommends actions.
- **Social media analytics:** Refers to doing a sentiment analysis to assess public opinion on topics or events. It also allows users to discover the behavior patterns and tastes of individuals, which can help industry target goods and services in a customized way.
- **Entity analytics:** This is a somewhat new area that groups data about entities of interest and learns more about them.
- **Cognitive computing:** Refers to an area of developing computing systems that will interact with people to give them better insight and advice.

In another book, Bill Franks of Teradata<sup>4</sup> voices a similar theme; he states that tapping big data for better analytics is essential for a competitive advantage in any industry today, and he shows how to develop a “big data advanced analytics ecosystem” in any organization to uncover new opportunities in business.

As we can see from all these industry-based publications by experts, big data is entering a new frontier in which big data will be harnessed to provide analytics-oriented applications that will lead to increased productivity, higher quality, and growth in all businesses. This chapter discusses the technology that has been created over the last decade to harness big data. We focus on those technologies that can be attributed to the MapReduce/Hadoop ecosystem, which covers most of the ground of open source projects for big data applications. We will not be able to get into the applications of the big data technology for analytics. That is a vast area by itself. Some of the basic data mining concepts are mentioned in Chapter 28; however, today’s analytics offerings go way beyond the basic concepts we have outlined there.

In Section 25.1, we introduce the essential features of big data. In Section 25.2, we will give the historical background behind the MapReduce/Hadoop technology and comment on the various releases of Hadoop. Section 25.3 discusses the underlying file system called Hadoop Distributed File System for Hadoop. We discuss its architecture, the I/O operations it supports, and its scalability. Section 25.4 provides further details on MapReduce (MR), including its runtime environment and high-level interfaces called Pig and Hive. We also show the power of MapReduce in terms of the relational join implemented in various ways. Section 25.5 is devoted to the later development called Hadoop v2 or MRv2 or YARN, which separates resource management from job management. Its rationale is explained first, and then its architecture and other frameworks being developed on YARN are explained. In Section 25.6 we discuss some general issues related to the MapReduce/Hadoop technology. First we discuss this technology vis-à-vis the parallel DBMS technology. Then we discuss it in the context of cloud computing, and we mention the data locality issues for improving performance. YARN as a data service platform is discussed next, followed by the challenges for big data technology in general. We end this chapter in Section 25.7 by mentioning some ongoing projects and summarizing the chapter.

---

<sup>4</sup>See Franks (2013) : *Taming The Big Data Tidal Wave*.

## 25.1 What Is Big Data?

*Big data* is becoming a popular and even a fashionable term. People use this term whenever a large amount of data is involved with some analysis; they think that using this term will make the analysis look like an advanced application. However, the term *big data* legitimately refers to datasets whose size is beyond the ability of typical database software tools to capture, store, manage, and analyze. In today's environment, the size of datasets that may be considered as big data ranges from terabytes ( $10^{12}$  bytes), or petabytes ( $10^{15}$  bytes), to exabytes ( $10^{18}$  bytes). The notion of what is Big data will depend on the industry, how data is used, how much historical data is involved and many other characteristics. The Gartner Group, a popular enterprise-level organization that industry looks up to for learning about trends, characterized big data in 2011 by the three V's: volume, velocity, and variety. Other characteristics, such as veracity and value, have been added to the definition by other researchers. Let us briefly see what these stand for.

**Volume.** The volume of data obviously refers to the size of data managed by the system. Data that is somewhat automatically generated tends to be voluminous. Examples include sensor data, such as the data in manufacturing or processing plants generated by sensors; data from scanning equipment, such as smart card and credit card readers; and data from measurement devices, such as smart meters or environmental recording devices.

The **industrial internet of things** (IIOT or IOT) is expected to bring about a revolution that will improve the operational efficiency of enterprises and open up new frontiers for harnessing intelligent technologies. The IOT will cause billions of devices to be connected to the Internet because these devices generate data continuously. For example, in gene sequencing, next generation sequencing (NGS) technology means that the volume of gene sequence data will be increased exponentially.

Many additional applications are being developed and are slowly becoming a reality. These applications include using remote sensing to detect underground sources of energy, environmental monitoring, traffic monitoring and regulation by automatic sensors mounted on vehicles and roads, remote monitoring of patients using special scanners and equipment, and tighter control and replenishment of inventories using radio-frequency identification (RFID) and other technologies. All these developments will have associated with them a large volume of data. Social networks such as Twitter and Facebook have hundreds of millions of subscribers worldwide who generate new data with every message they send or post they make. Twitter hit a half billion tweets daily in October 2012.<sup>5</sup> The amount of data required to store one second of high-definition video may equal 2,000 pages of text data. Thus, the multimedia data being uploaded on YouTube and similar video hosting platforms is significantly more voluminous than simple numeric or text data. In 2010, enterprises stored over 13 exabytes ( $10^{18}$  bytes) of data, which amounts to over 50,000 times the amount of data stored by the Library of Congress.<sup>6</sup>

<sup>5</sup>See Terdiman (2012): <http://www.cnet.com/news/report-twitter-hits-half-a-billion-tweets-a-day/>

<sup>6</sup>From Jagadish et al. (2014).

**Velocity.** The definition of *big data* goes beyond the dimension of volume; it includes the types and frequency of data that are disruptive to traditional database management tools. The McKinsey report on big data<sup>7</sup> described velocity as the speed at which data is created, accumulated, ingested, and processed. High velocity is attributed to data when we consider the typical speed of transactions on stock exchanges; this speed reaches billions of transactions per day on certain days. If we must process these transactions to detect potential fraud or we must process billions of call records on cell phones daily to detect malicious activity, we face the velocity dimension. Real-time data and streaming data are accumulated by the likes of Twitter and Facebook at a very high velocity. Velocity is helpful in detecting trends among people that are tweeting a million tweets every three minutes. Processing of streaming data for analysis also involves the velocity dimension.

**Variety.** Sources of data in traditional applications were mainly transactions involving financial, insurance, travel, healthcare, retail industries, and governmental and judicial processing. The types of sources have expanded dramatically and include Internet data (e.g., clickstream and social media), research data (e.g., surveys and industry reports), location data (e.g., mobile device data and geospatial data), images (e.g., surveillance, satellites and medical scanning), e-mails, supply chain data (e.g., EDI—electronic data interchange, vendor catalogs), signal data (e.g., sensors and RFID devices), and videos (YouTube enters hundreds of minutes of video every minute). Big data includes structured, semistructured, and unstructured data (see discussion in Chapter 26) in different proportions based on context.

Structured data feature a formally structured data model, such as the relational model, in which data are in the form of tables containing rows and columns, and a hierarchical database in IMS, which features record types as segments and fields within a record.

Unstructured data have no identifiable formal structure. We discussed systems like MongoDB (in Chapter 24), which stores unstructured document-oriented data, and Neo4j, which stores data in the form of a graph. Other forms of unstructured data include e-mails and blogs, PDF files, audio, video, images, clickstreams, and Web contents. The advent of the World Wide Web in 1993–1994 led to tremendous growth in unstructured data. Some forms of unstructured data may fit into a format that allows well-defined tags that separate semantic elements; this format may include the capability to enforce hierarchies within the data. XML is hierarchical in its descriptive mechanism, and various forms of XML have come about in many domains; for example, biology (bioML—biopolymer markup language), GIS (gML—geography markup language), and brewing (BeerXML—language for exchange of brewing data), to name a few. Unstructured data constitutes the major challenge in today's big data systems.

**Veracity.** The veracity dimension of big data is a more recent addition than the advent of the Internet. Veracity has two built-in features: the credibility of the source, and the suitability of data for its target audience. It is closely related to trust;

---

<sup>7</sup>See McKinsey (2013).

listing veracity as one of the dimensions of big data amounts to saying that data coming into the so-called big data applications have a variety of trustworthiness, and therefore before we accept the data for analytical or other applications, it must go through some degree of quality testing and credibility analysis. Many sources of data generate data that is uncertain, incomplete, and inaccurate, therefore making its veracity questionable.

We now turn our attention to the technologies that are considered the pillars of big data technologies. It is anticipated that by 2016, more than half of the data in the world may be processed by Hadoop-related technologies. It is therefore important for us to trace the MapReduce/Hadoop revolution and understand how this technology is positioned today. The historical development starts with the programming paradigm called MapReduce programming.

## 25.2 Introduction to MapReduce and Hadoop

In this section, we will introduce the technology for big data analytics and data processing known as Hadoop, an open source implementation of the MapReduce programming model. The two core components of Hadoop are the MapReduce programming paradigm and HDFS, the Hadoop Distributed File System. We will briefly explain the background behind Hadoop and then MapReduce. Then we will make some brief remarks about the Hadoop ecosystem and the Hadoop releases.

### 25.2.1 Historical Background

Hadoop has originated from the quest for an open source search engine. The first attempt was made by the then Internet archive director Doug Cutting and University of Washington graduate student Mike Carafella. Cutting and Carafella developed a system called Nutch that could crawl and index hundreds of millions of Web pages. It is an open source Apache project.<sup>8</sup> After Google released the Google File System<sup>9</sup> paper in October 2003 and the MapReduce programming paradigm paper<sup>10</sup> in December 2004, Cutting and Carafella realized that a number of things they were doing could be improved based on the ideas in these two papers. They built an underlying file system and a processing framework that came to be known as Hadoop (which used Java as opposed to the C++ used in MapReduce) and ported Nutch on top of it. In 2006, Cutting joined Yahoo, where there was an effort under way to build open source technologies using ideas from the Google File System and the MapReduce programming paradigm. Yahoo wanted to enhance its search processing and build an open source infrastructure based on the Google File System and MapReduce. Yahoo spun off the storage engine and the processing parts of Nutch as **Hadoop** (named after the stuffed elephant toy of Cutting's son). The

---

<sup>8</sup>For documentation on Nutch, see <http://nutch.apache.org>

<sup>9</sup>Ghemawat, Gbioff, and Leung (2003).

<sup>10</sup>Dean and Ghemawat (2004).



initial requirements for Hadoop were to run batch processing using cases with a high degree of scalability. However, the circa 2006 Hadoop could only run on a handful of nodes. Later, Yahoo set up a research forum for the company's data scientists; doing so improved the search relevance and ad revenue of the search engine and at the same time helped to mature the Hadoop technology. In 2011, Yahoo spun off Hortonworks as a Hadoop-centered software company. By then, Yahoo's infrastructure contained hundreds of petabytes of storage and 42,000 nodes in the cluster. In the years since Hadoop became an open source Apache project, thousands of developers worldwide have contributed to it. A joint effort by Google, IBM, and NSF used a 2,000-node Hadoop cluster at a Seattle data center and helped further universities' research on Hadoop. Hadoop has seen tremendous growth since the 2008 launch of Cloudera as the first commercial Hadoop company and the subsequent mushrooming of a large number of startups. IDC, a software industry market analysis firm, predicts that the Hadoop market will surpass \$800 million in 2016; IDC predicts that the big data market will hit \$23 billion in 2016. For more details about the history of Hadoop, consult a four-part article by Harris.<sup>11</sup>

An integral part of Hadoop is the MapReduce programming framework. Before we go any further, let us try to understand what the MapReduce programming paradigm is all about. We defer a detailed discussion of the HDFS file system to Section 25.3.

### 25.2.2 MapReduce

The MapReduce programming model and runtime environment was first described by Jeffrey Dean and Sanjay Ghemawat (Dean & Ghemawat (2004)) based on their work at Google. Users write their programs in a functional style of *map* and *reduce* tasks, which are automatically parallelized and executed on large clusters of commodity hardware. The programming paradigm has existed as far back as the language LISP, which was designed by John McCarthy in late 1950s. However, the reincarnation of this way of doing parallel programming and the way this paradigm was implemented at Google gave rise to a new wave of thinking that contributed to the subsequent developments of technologies such as Hadoop. The runtime system handles many of the messy engineering aspects of parallelization, fault tolerance, data distribution, load balancing, and management of task communication. As long as users adhere to the **contracts** laid out by the MapReduce system, they can just focus on the logical aspects of this program; this allows programmers without distributed systems experience to perform analysis on very large datasets.

The motivation behind the MapReduce system was the years spent by the authors and others at Google implementing hundreds of special-purpose computations on large datasets (e.g., computing inverted indexes from Web content collected via Web crawling; building Web graphs; and extracting statistics from Web logs, such as frequency distribution of search requests by topic, by region, by type of user, etc.). Conceptually, these tasks are not difficult to express; however, given the scale

---

<sup>11</sup>Derreck Harris : 'The history of Hadoop: from 4 nodes to the future of data,' at <https://gigaom.com/2013/03/04/the-history-of-hadoop-from-4-nodes-to-the-future-of-data/>



of data in billions of Web pages and with the data spread over thousands of machines, the execution task was nontrivial. Issues of program control and data management, data distribution, parallelization of computation, and handling of failures became critically important.

The MapReduce programming model and runtime environment was designed to cope with the above complexity. The abstraction is inspired by the map and reduce primitives present in LISP and many other functional languages. An underlying model of data is assumed; this model treats an object of interest in the form of a unique key that has associated content or value. This is the key-value pair. Surprisingly, many computations can be expressed as applying a map operation to each logical “record” that produces a set of intermediate key-value pairs and then applying a reduce operation to all the values that shared the same key (the purpose of sharing is to combine the derived data). This model allows the infrastructure to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance. The idea of providing a restricted programming model so that the runtime can parallelize computations automatically is not new. MapReduce is the enhancement of those existing ideas. As it is understood today, MapReduce is a fault-tolerant implementation and a runtime environment that scales to thousands of processors. The programmer is spared the worry of handling failures. In subsequent sections, we will abbreviate MapReduce as **MR**.

**The MapReduce Programming Model** In the following description, we use the formalism and description as it was originally described by Dean and Ghemawat (2010).<sup>12</sup> The map and reduce functions have the following general form:

$$\begin{aligned} \text{map}[K1, V1] \text{ which is } (key, value) : \text{List}[K2, V2] \text{ and} \\ \text{reduce}(K2, \text{List}[V2]) : \text{List}[K3, V3] \end{aligned}$$

**Map** is a generic function that takes a key of type **K1** and a value of type **V1** and returns a list of key-value pairs of type **K2** and **V2**. **Reduce** is a generic function that takes a key of type **K2** and a list of values of type **V2** and returns pairs of type **(K3, V3)**. In general, the types K1, K2, K3, etc., are different, with the only requirement that the output types from the Map function must match the input type of the Reduce function.

The basic execution workflow of MapReduce is shown in Figure 25.1.

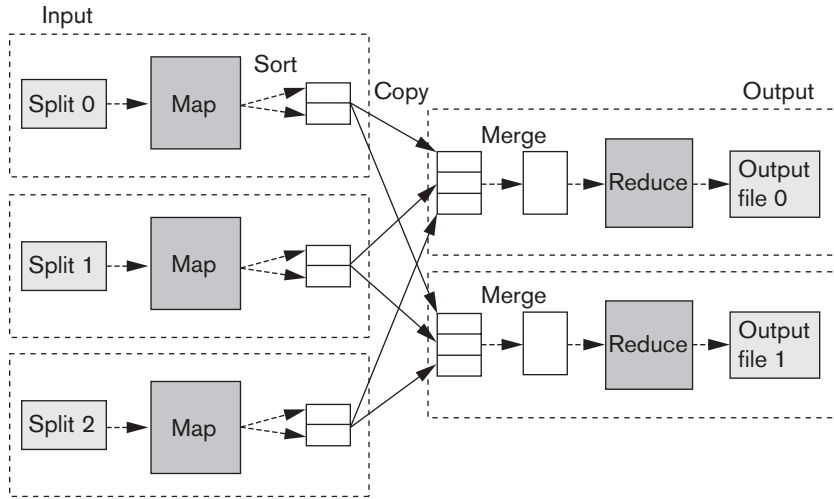
Assume that we have a document and we want to make a list of words in it with their corresponding frequencies. This ubiquitous *word count* example quoted directly from Dean and Ghemawat (2004) above goes as follows in pseudocode:

**Map (String key, String value):**  
**for each word w in value Emitintermediate (w, “1”);**

Here key is the document name, and value is the text content of the document.

---

<sup>12</sup>Jeffrey Dean and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in OSDI (2004).



**Figure 25.1**  
Overview of MapReduce  
execution. (Adapted  
from T. White, 2012)

Then the above lists of (word, 1) pairs are added up to output total counts of all words found in the document as follows:

```
Reduce (String key, Iterator values) : // here the key is a word and values are  
lists of its counts //  
  Int result =0;  
  For each v in values :  
    result += Parseint (v);  
  Emit (key, Asstring (result));
```

The above example in MapReduce programming appears as:

```
map[LongWritable,Text](key, value) : List[Text, LongWritable] = {  
  String[] words = split(value)  
  for(word : words) {  
    context.out(Text(word), LongWritable(1))  
  }  
}  
reduce[Text, Iterable[LongWritable]](key, values) : List[Text, LongWritable] = {  
  LongWritable c = 0  
  for( v : values) {  
    c += v  
  }  
  context.out(key,c)  
}
```

The data types used in the above example are LongWritable and Text. Each MapReduce job must register a Map and Reduce function. The Map function receives each key-value pair and on each call can output 0 or more key-value pairs. The signature of the Map function specifies the data types of its input and output

key-value pairs. The Reduce function receives a key and an iterator of values associated with that key. It can output one or more key-value pairs on each invocation. Again, the signature of the Reduce function indicates the data types of its inputs and outputs. The output type of the Map must match the input type of the Reduce function. In the wordcount example, the map function receives each line as a value, splits it into words, and emits (via the function context.out) a row for each word with frequency 1. Each invocation of the Reduce function receives for a given word the list of frequencies computed on the Map side. It adds these and emits each word and its frequency as output. The functions interact with a *context*. The context is used to interact with the framework. It is used by clients to send configuration information to tasks; and tasks can use it to get access to HDFS and read data directly from HDFS, to output key-value pairs, and to send status (e.g., task counters) back to the client.

The MapReduce way of implementing some other functions based on Dean and Ghemawat (2004) is as follows:

### **Distributed Grep**

Grep looks for a given pattern in a file. The Map function emits a line if it matches a supplied pattern. The Reduce function is an identity function that copies the supplied intermediate data to the output. This is an example of a *Map only task*; there is no need to incur the cost of a **Shuffle**. We will provide more information when we explain the MapReduce runtime.

### **Reverse Web-Link Graph**

The purpose here is to output (target URL, source URL) pairs for each link to a target page found in a page named source. The Reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair <target, list(source)>.

### **Inverted Index**

The purpose is to build an inverted index based on all words present in a document repository. The Map function parses each document and emits a sequence of (word, document\_id) pairs. The Reduce function takes all pairs for a given word, sorts them by document\_id and emits a (word, list (document\_id)) pair. The set of all these pairs forms an inverted index.

These illustrative applications give a sense of the MapReduce programming model's broad applicability and the ease of expressing the application's logic using the Map and Reduce phases.

A **Job** in MapReduce comprises the code for the Map and Reduce (usually packaged as a jar) phases, a set of artifacts needed to run the tasks (such as files, other jars, and archives) and, most importantly, a set of properties specified in a configuration. There are hundreds of properties that can be specified, but the core ones are as follows:

- the Map task
- the Reduce task

- the Input that the Job is to run on: typically specified as an HDFS path(s)
- the Format(Structure) of the Input
- the Output path
- the Output Structure
- the Reduce-side parallelism

A Job is submitted to the **JobTracker**, which then schedules and manages the execution of the Job. It provides a set of interfaces to monitor running Jobs. See the Hadoop Wiki<sup>13</sup> for further details about the workings of the JobTracker.

### 25.2.3 Hadoop Releases

Since the advent of Hadoop as a new distributed framework to run MapReduce programs, various releases have been produced:

The 1.x releases of Hadoop are a continuation of the original 0.20 code base. Subreleases with this line have added Security, additional HDFS and MapReduce improvements to support HBase, a better MR programming model, as well as other improvements.

The 2.x releases include the following major features:

- YARN (Yet Another Resource Navigator) is a general resource manager extracted out of the JobTracker from MR version1.
- A new MR runtime that runs on top of YARN.
- Improved HDFS that supports federation and increased availability.

At the time of this writing, Hadoop 2.0 has been around for about a year. The adoption is rapidly picking up; but a significant percentage of Hadoop deployments still run on Hadoop v1.

## 25.3 Hadoop Distributed File System (HDFS)

As we said earlier, in addition to MapReduce, the other core component of Hadoop is the underlying file system HDFS. In this section, we will first explain the architecture of HDFS, then describe the file input/output operations supported in HDFS, and finally comment on the scalability of HDFS.

### 25.3.1 HDFS Preliminaries

The Hadoop Distributed File System (HDFS) is the file system component of Hadoop and is designed to run on a cluster of commodity hardware. HDFS is patterned after the UNIX file system; however, it relaxes a few POSIX (portable operating system interface) requirements to enable streaming access to file system data. HDFS provides high-throughput access to large datasets. HDFS stores file system

---

<sup>13</sup>Hadoop Wiki is at <http://hadoop.apache.org/>

metadata and application data separately. Whereas the metadata is stored on a dedicated server, called the **NameNode**, the application data is stored on other servers, called **DataNodes**. All servers are fully connected and communicate with each other using TCP-based protocols. To make data durable, the file content is replicated on multiple **DataNodes**, as in the Google File System. This not only increases reliability, but it also multiplies the bandwidth for data transfer and enables colocation of computation with data. It was designed with the following assumptions and goals:

**Hardware failure:** Using commodity hardware, failure of hardware is the norm rather than an exception. Therefore, with thousands of nodes, automatic detection and recovery from failures becomes a must.

**Batch processing:** HDFS has been primarily designed for batch rather than interactive use. High throughput is emphasized over low latency of data access. Full scans of files are typical.

**Large datasets:** HDFS was designed to support huge files in the hundreds of gigabytes to terabytes range.

**Simple coherency model:** HDFS applications need a one writer and many reader access models for files. File content cannot be updated, but only appended. This model alleviates coherency issues among copies of data.

### 25.3.2 Architecture of HDFS

HDFS has a master-slave architecture. The master server, called the **NameNode**, manages the file system storage area or namespace; Clients access the namespace through the **Namenode**. The slaves called **DataNodes** run on a cluster of commodity machines, usually one per machine. They manage the storage attached to the node that they run on. The namespace itself comprises Files and Directories. The **Namenodes** maintain *inodes* (index nodes) about File and Directories with attributes like ownership, permissions, creation and access times, and disk space quotas. Using *inodes*, the mapping of File blocks to **DataNodes** is determined. **DataNodes** are responsible for serving read and write requests from clients. **DataNodes** perform block creation, deletion, and replication operations as instructed by the **NameNode**. A cluster can have thousands of **DataNodes** and tens of thousands of HDFS clients simultaneously connected.

To read a file, a client first connects to the **NameNode** and obtains the locations of the data blocks in the file it wants to access; it then connects directly with the **DataNodes** that house the blocks and reads the data.

The architecture of HDFS has the following highlights:

1. HDFS allows a decoupling of metadata from data operations. Metadata operations are fast whereas data transfers are much slower. If the location of metadata and transfer of data are not decoupled, speed suffers in a distributed environment because data transfer dominates and slows the response.

2. Replication is used to provide reliability and high availability. Each block is replicated (default is three copies) to a number of nodes in the cluster. The highly contentious files like MapReduce job libraries would have a higher number of replicas to reduce network traffic.
3. The network traffic is kept to a minimum. For reads, clients are directed to the closest DataNode. As far as possible, a local file system read is attempted and involves no network traffic; the next choice is a copy on a node on the same rack before going to another rack. For writes, to reduce network bandwidth utilization, the first copy is written to the same node as the client. For other copies, travel across racks is minimized.

**NameNode.** The NameNode maintains an **image** of the file system comprising *i*-nodes and corresponding block locations. Changes to the file system are maintained in a Write-ahead commit log (see the discussion of Write-ahead logs in Chapter 22) called the **Journal**. Checkpoints are taken for purposes of recovery; they represent a persistent record of the image without the dynamic information related to the block placement. Block placement information is obtained from the DataNodes periodically as described below. During Restart, the image is restored to the last checkpoint and the journal entries are applied to that image. A new checkpoint and empty journal are created so that the NameNode can start accepting new client requests. The startup time of a NameNode is proportional to the Journal file's size. Merging the checkpoint with the Journal periodically reduces restart time.

Note that with the above architecture, it is catastrophic to have any corruption of the Checkpoint or the Journal. To guard against corruption, both are written to multiple directories on different volumes.

**Secondary NameNodes.** These are additional NameNodes that can be created to perform either the checkpointing role or a backup role. A Checkpoint node periodically combines existing checkpoint and journal files. In backup mode, it acts like another storage location for the Journal for the primary NameNode. The backup NameNode remains up-to-date with the file system and can take over on failure. In Hadoop V1, this takeover must be done manually.

**DataNodes:** Blocks are stored on a DataNode in the node's native file system. The NameNode directs clients to the DataNodes that contain a copy of the block they want to read. Each block has its representation in two files in the native file system: a file containing the data and a second file containing the metadata, which includes the checksums for the block data and the block's generation stamp. DataNodes and NameNodes do not communicate directly but via a so-called **heartbeat mechanism**, which refers to a periodic reporting of the state by the DataNode to the NameNode; the report is called a Block Report. The report contains the block id, the generation stamp, and the length for each block. The block locations are not part of the namespace image. They must be obtained from the block reports, and they change as blocks are moved around. The MapReduce Job Tracker, along with the

NameNode, uses the latest block report information for scheduling purposes. In response to a heartbeat from the DataNode, the NameNode sends one of the following types of commands to the DataNode:

- Replicate a block to another node.
- Remove a block replica.
- Reregister the node or shut down the node.
- Send an immediate block report.

### 25.3.3 File I/O Operations and Replica Management in HDFS

HDFS provides a single-writer, multiple-reader model. Files cannot be updated, but only appended. A file consists of blocks. Data is written in 64-KB packets in a **write pipeline**, which is set up to minimize network utilization, as we described above. Data written to the last block becomes available only after an explicit hflush operation. Simultaneous reading by clients is possible while data is being written. A checksum is generated and stored for each block and is verified by the client to detect corruption of data. Upon detection of a corrupt block, the Namenode is notified; it initiates a process to replicate the block and instructs the Datanode to remove the corrupt block. During the read operation, an attempt is made to fetch a replica from as close a node as possible by ordering the nodes in ascending order of distance from the client. A read fails when the Datanode is unavailable, when the checksum test fails, or when the replica is no longer on the Datanode. HDFS has been optimized for batch processing similar to MapReduce.

**Block Placement.** Nodes of a Hadoop cluster are typically spread across many racks. They are normally organized such that nodes on a rack share a switch, and rack switches are connected to a high-speed switch at the upper level. For example, the rack level may have a 1-Gb switch, whereas at the top level there may be a 10-Gb switch. HDFS estimates the network bandwidth between Datanodes based on their distance. Datanodes on the same physical node have a distance of 0, on the same rack are distance 2 away, and on different racks are distance 4 away. The default HDFS block placement policy balances between minimizing the write cost and maximizing data reliability and availability as well as aggregate read bandwidth. Network bandwidth consumed is estimated based on distance among DataNodes. Thus, for DataNodes on the same physical node, the distance is 0, whereas on the same rack it is 2 and on a different rack it is 4. The ultimate goal of block placement is to minimize the write cost while maximizing data availability and reliability as well as available bandwidth for reading. Replicas are managed so that there is at least one on the original node of the client that created it, and others are distributed among other racks. Tasks are preferred to be run on nodes where the data resides; three replicas gives the scheduler enough leeway to place tasks where the data is.

**Replica Management.** Based on the block reports from the DataNodes, the NameNode tracks the number of replicas and the location of each block. A replication priority queue contains blocks that need to be replicated. A background thread



monitors this queue and instructs a DataNode to create replicas and distribute them across racks. NameNode prefers to have as many different racks as possible to host replicas of a block. Overreplicated blocks cause some replicas to be removed based on space utilization of the DataNodes.

### 25.3.4 HDFS Scalability

Since we are discussing big data technologies in this chapter, it is apropos to discuss some limits of scalability in HDFS. Hadoop program management committee member Shvachko commented that the Yahoo HDFS cluster had achieved the following levels as opposed to the intended targets (Shvachko, 2010). The numbers in parentheses are the targets he listed. Capacity: 14 petabytes (vs. 10 petabytes); number of nodes: 4,000 (vs. 10,000); clients: 15,000 (vs. 100,000); and files: 60 million (vs. 100 million). Thus, Yahoo had come very close to its intended targets in 2010, with a smaller cluster of 4,000 nodes and fewer clients; but Yahoo had actually exceeded the target with respect to total amount of data handled.

Some of the observations made by Shvachko (2010) are worth mentioning. They are based on the HDFS configuration used at Yahoo in 2010. We present the actual and estimated numbers below to give the reader a sense of what is involved in these gigantic data processing environments.

- The blocksize used was 128K, and an average file contained 1.5 blocks. NameNode used about 200 bytes per block and an additional 200 bytes for an *i*-node. 100 million files referencing 200 million blocks would require RAM capacity exceeding 60 GB.
- For 100 million files with size of 200 million blocks and a replication factor of 3, the disk space required is 60 PB. Thus a rule of thumb was proposed that 1 GB of RAM in NameNode roughly corresponds to 1 PB of data storage based on the assumption of 128K blocksize and 1.5 blocks per file.
- In order to hold 60 PB of data on a 10,000-node cluster, each node needs a capacity of 6 TB. This can be achieved by having eight 0.75-TB drives.
- The internal workload for the NameNode is block reports. About 3 reports per second containing block information on 60K blocks per report were received by the NameNode.
- The external load on the NameNode consisted of external connections and tasks from MapReduce jobs. This resulted in tens of thousands of simultaneous connections.
- The Client Read consisted of performing a block lookup to get block locations from the NameNode, followed by accessing the nearest replica of the block. A typical client (the Map job from an MR task) would read data from 1,000 files with an average reading of half a file each, amounting to 96 MB of data. This was estimated to take 1.45 seconds. At that rate, 100,000 clients would send 68,750 block-location requests per second to the NameNode. This was considered to be well within the capacity of the NameNode, which was rated at handling 126K requests per second.



- The write workload: Given a write throughput of 40 MB/sec, an average client writes 96 MB in 2.4 sec. That creates over 41K “create block” requests from 100,000 nodes at the NameNode. This was considered far above the NameNode capacity.

The above analysis assumed that there was only one task per node. In reality, there could be multiple tasks per node as in the real system at Yahoo, which ran 4 MapReduce (MR) tasks per node. The net result was a bottleneck at the NameNode. Issues such as these have been handled in Hadoop v2, which we discuss in the next section.

### 25.3.5 The Hadoop Ecosystem

Hadoop is best known for the MapReduce programming model, its runtime infrastructure, and the Hadoop Distributed File System (HDFS). However, the Hadoop ecosystem has a set of related projects that provide additional functionality on top of these core projects. Many of them are top-level open source Apache projects and have a very large contributing user community of their own. We list a few important ones here:

**Pig and Hive:** These provide a higher level interface for working with the Hadoop framework.

- Pig provides a dataflow language. A script written in PigScript translates into a directed acyclic graph (DAG) of MapReduce jobs.
- Hive provides an SQL interface on top of MapReduce. Hive’s SQL support includes most of the SQL-92 features and many of the advanced analytics features from later SQL standards. Hive also defines the SerDe (Serialization/ Deserialization) abstraction, which defines a way of modeling the record structure on datasets in HDFS beyond just key-value pairs. We will discuss both of these in detail in Section 25.4.4.

**Oozie:** This is a service for scheduling and running workflows of Jobs; individual steps can be MR jobs, Hive queries, Pig scripts, and so on.

**Sqoop:** This is a library and a runtime environment for efficiently moving data between relational databases and HDFS.

**HBase:** This is a column-oriented key-value store that uses HDFS as its underlying store. (See Chapter 24 for a more detailed discussion of HBase.) It supports both batch processing using MR and key-based lookups. With proper design of the key-value scheme, a variety of applications are implemented using HBase. They include time series analysis, data warehousing, generation of cubes and multi-dimensional lookups, and data streaming.

## 25.4 MapReduce: Additional Details

We introduced the MapReduce paradigm in Section 25.2.2. We now elaborate further on it in terms of the MapReduce runtime. We discuss how the relational operation of join can be handled using MapReduce. We examine the high-level interfaces of Pig and Hive. Finally, we discuss the advantages of the combined MapReduce/Hadoop.

### 25.4.1 MapReduce Runtime

The purpose of this section is to give a broad overview of the MapReduce runtime environment. For a detailed description, the reader is encouraged to consult White (2012). MapReduce is a master-slave system that usually runs on the same cluster as HDFS. Typically, medium to large Hadoop clusters consist of a two- or three-level architecture built with rack-mounted servers.

**JobTracker.** The master process is called the *JobTracker*. It is responsible for managing the life cycle of Jobs and scheduling Tasks on the cluster. It is responsible for:

- Job submission, initializing a Job, providing Job status and state to both clients and TaskTrackers (the slaves), and Job completion.
- Scheduling Map and Reduce tasks on the cluster. It does this using a pluggable Scheduler.

**TaskTracker.** The slave process is called a *TaskTracker*. There is one running on all **Worker nodes** of the cluster. The Map-Reduce tasks run on Worker nodes. TaskTracker daemons running on these nodes register with the JobTracker on startup. They run tasks that the JobTracker assigns to them. Tasks are run in a separate process on the node; the life cycle of the process is managed by the TaskTracker. The TaskTracker creates the task process, monitors its execution, sends periodic status heartbeats to the JobTracker, and under failure conditions can kill the process at the request of the JobTracker. The TaskTracker provides services to the Tasks, the most important of which is the **Shuffle**, which we describe in a subsection below.

#### A. Overall flow of a MapReduce Job

A MapReduce job goes through the processes of Job Submission, Job Initialization, Task Assignment, Task Execution, and finally Job Completion. The Job Tracker and Task Tracker we described above are both involved in these. We briefly review them below.

**Job submission** A client submits a Job to the **JobTracker**. The Job package contains the executables (as a jar), any other components (files, jars archives) needed to execute the Job, and the InputSplits for the Job.

**Job initialization** The JobTracker accepts the Job and places it on a Job Queue. Based on the input splits, it creates map tasks for each split. A number of reduce tasks are created based on the Job configuration.

**Task assignment** The JobTracker's scheduler assigns Task to the TaskTracker from one of the running Jobs. In Hadoop v1, TaskTrackers have a fixed number of slots for map tasks and for reduce tasks. The Scheduler takes the location information of the input files into account when scheduling tasks on cluster nodes.

**Task execution** Once a task has been scheduled on a slot, the TaskTracker manages the execution of the task: making all Task artifacts available to the

Task process, launching the Task JVM, monitoring the process and coordinating with the JobTracker to perform management operations like cleanup on Task exit, and killing Tasks on failure conditions. The TaskTracker also provides the *Shuffle Service* to Tasks; we describe this when we discuss the Shuffle Procedure below.

**Job completion** Once the last Task in a Job is completed, the JobTracker runs the Job cleanup task (which is used to clean up intermediate files in both HDFS and the local file systems of TaskTrackers).

## B. Fault Tolerance in MapReduce

There are three kinds of failures: failure of the Task, failure of the TaskTracker, and failure of the JobTracker.

**Task failure** This can occur if the Task code throws a Runtime exception, or if the Java Virtual Machine crashes unexpectedly. Another issue is when the TaskTracker does not receive any updates from the Task process for a while (the time period is configurable). In all these cases the TaskTracker notifies the JobTracker that the Task has failed. When the JobTracker is notified of the failure, it will reschedule execution of the task.

**TaskTracker failure** A TaskTracker process may crash or become disconnected from the JobTracker. Once the JobTracker marks a TaskTracker as failed, any map tasks completed by the TaskTracker are put back on the queue to be rescheduled. Similarly, any map task or reduce task in progress on a failed TaskTracker is also rescheduled.

**JobTracker failure** In Hadoop v1, JobTracker failure is not a recoverable failure. The JobTracker is a Single Point of Failure. The JobTracker has to be manually restarted. On restart all the running jobs have to be resubmitted. This is one of the drawbacks of Hadoop v1 that have been addressed by the next generation of Hadoop MapReduce called YARN.

**Semantics in the presence of failure** When the user-supplied map and reduce operators are deterministic functions of their input values, the MapReduce system produces the same output as would have been produced by a nonfaulting sequential execution of the entire program. Each task writes its output to a private task directory. If the JobTracker receives multiple completions for the same Task, it ignores all but the first one. When a Job is completed, Task outputs are moved to the Job output directory.

## C. The Shuffle Procedure

A key feature of the MapReduce (MR) programming model is that the reducers get all the rows for a given key together. This is delivered by what is called the MR **shuffle**. The shuffle is divided into the Map, Copy, and Reduce phases.

**Map phase:** When rows are processed in Map tasks, they are initially held in an in-memory buffer, the size of which is configurable (the default is 100 MB). A

background thread partitions the buffered rows based on the number of Reducers in the job and the *Partitioner*. The *Partitioner* is a pluggable interface that is asked to choose a Reducer for a given Key value and the number of reducers in the Job. The partitioned rows are sorted on their key values. They can further be sorted on a provided *Comparator* so that rows with the same key have a stable sort order. This is used for Joins to ensure that for rows with the same key value, rows from the same table are bunched together. Another interface that can be plugged in is the *Combiner* interface. This is used to reduce the number of rows output per key from a mapper and is done by applying a reduce operation on each Mapper for all rows with the same key. During the Map phase, several iterations of partitioning, sorting, and combining may happen. The end result is a single local file per reducer that is sorted on the Key.

**Copy phase:** The Reducers pull their files from all the Mappers as they become available. These are provided by the JobTracker in Heartbeat responses. Each Mapper has a set of listener threads that service Reducer requests for these files.

**Reduce phase:** The Reducer reads all its files from the Mappers. All files are merged before streaming them to the Reduce function. There may be multiple stages of merging, depending on how the Mapper files become available. The Reducer will avoid unnecessary merges; for example, the last N files will be merged as the rows are being streamed to the Reduce function.

## D. Job Scheduling

The **JobTracker** in MR 1.0 is responsible for scheduling work on cluster nodes. Clients' submitted jobs are added to the Job Queue of the JobTracker. The initial versions of Hadoop used a FIFO scheduler that scheduled jobs sequentially as they were submitted. At any given time, the cluster would run the tasks of a single Job. This caused undue delays for short jobs like ad-hoc hive queries if they had to wait for long-running machine learning-type jobs. The wait times would exceed runtimes, and the throughput on the cluster would suffer. Additionally, the cluster also would remain underutilized. We briefly describe two other types of schedulers, called the Fair Scheduler and Capacity Scheduler, that alleviate this situation.

**Fair Scheduler:** The goal of Fair Scheduler is to provide fast response time to small jobs in a Hadoop shared cluster. For this scheduler, jobs are grouped into Pools. The capacity of the cluster is evenly shared among the Pools. At any given time the resources of the cluster are evenly divided among the Pools, thereby utilizing the capacity of the cluster evenly. A typical way to set up Pools is to assign each user a Pool and assign certain Pools a minimum number of slots.

**Capacity Scheduler:** The Capacity Scheduler is geared to meet the needs of large Enterprise customers. It is designed to allow multiple tenants to share resources of a large Hadoop cluster by allocating resources in a timely manner under a given set of capacity constraints. In large enterprises, individual departments are apprehensive of using one centralized Hadoop cluster for concerns

that they may not be able to meet the service-level agreements (SLAs) of their applications. The Capacity Scheduler is designed to give each tenant guarantees about cluster capacity using the following provisions:

- ❑ There is support for multiple queues, with hard and soft limits in terms of fraction of resources.
- ❑ Access control lists (ACLs) are used that determine who can submit, view, and modify the Jobs in a queue.
- ❑ Excess capacity is evenly distributed among active Queues.
- ❑ Tenants have usage limits; such limits prevent tenants from monopolizing the cluster.

### 25.4.2 Example: Achieving Joins in MapReduce

To understand the power and utility of the MapReduce programming model, it is instructive to consider the most important operation of relational algebra, called Join, which we introduced in Chapter 6. We discussed its use via SQL queries (Chapters 7 and 8) and its optimization (Chapters 18 and 19). Let us consider the problem of joining two relations  $R(A, B)$  with  $S(B, C)$  with the join condition  $R.A = S.B$ . Assume both tables reside on HDFS. Here we list the many strategies that have been devised to do equi-joins in the MapReduce environment.

**Sort-Merge Join.** The broadest strategy for performing a join is to utilize the Shuffle to partition and sort the data and have the reducers merge and generate the output. We can set up an MR job that reads blocks from both tables in the Map phase. We set up a *Partitioner* to hash partition rows from  $R$  and  $S$  on the value of the  $B$  column. The key output from the Map phase includes a table *tag*. So the key has the form (tag, (key)). In MR, we can configure a custom Sort for the Job's shuffle; the custom Sort sorts the rows that have the same key. In this case, we Sort rows with the same  $B$  value based on the tag. We give the smaller table a tag of 0 and the larger table a tag of 1. So a reducer will see all rows with the same  $B$  value in the order: smaller table rows first, then larger table rows. The Reducer can buffer smaller table rows; once it starts to receive large table rows, it can do an in-memory cross-product with the buffered small table rows to generate the join output. The cost of this strategy is dominated by the shuffle cost, which will write and read each row multiple times.

**Map-Side Hash Join.** For the case when one of  $R$  or  $S$  is a small table that can be loaded in the memory of each task, we can have the Map phase operate only on the large table splits. Each Map task can read the entire small table and create an in-memory hash map based on  $B$  as the hash key. Then it can perform a hash join. This is similar to Hash Joins in databases. The cost of this task is roughly the cost of reading the large table.

**Partition Join.** Assume that both  $R$  and  $S$  are stored in such a way that they are partitioned on the join keys. Then all rows in each Split belong to a certain identifiable range of the domain of the join field, which is  $B$  in our example. Assume both  $R$  and  $S$  are stored as  $p$  files. Suppose file ( $i$ ) contains rows such that  $(\text{Value } B) \bmod$

$p = i$ . Then we only need to join the  $i$ th file of  $\setminus(R)$   $R$  with the corresponding  $i$ th file of  $S$ . One way to do this is to perform a variation of the Map-Side join we discussed above: have the Mapper handling the  $i$ th partition of the larger table read the  $i$ th partition from the smaller table. This strategy can be expanded to work even when the two tables do not have the same number of partitions. It is sufficient for one to be a multiple of the other. For example, if table  $A$  is divided into two partitions and table  $B$  is divided into four partitions, then partition 1 from table  $A$  needs to join with partitions 1 and 3 of  $B$ , and partition 2 of  $A$  needs to join with partitions 2 and 4 of  $B$ . The opportunity to perform Bucketed Join (see below) is also common: for example, assume  $R$  and  $S$  are outputs of previous sort-merge joins. The output of the sort-merge join is partitioned in the joining expressions. Further joining this dataset allows us to avoid a shuffle.

**Bucket Joins.** This is a combination of Map-Side and Partition Joins. In this case only one relation, say the right side relation, is Partitioned. We can then run Mappers on the left side relation and perform a Map Join against each Partition from the right side.

**$N$ -Way Map-Side Joins.** A join on  $R(A, B, C, D)$ ,  $S(B, E)$ , and  $T(C, F)$  can be achieved in one MR job provided the rows for a key for all small tables can be buffered in memory. The join is typical in Data Warehouses (see Chapter 29), where  $R$  is a fact table and  $S$  and  $T$  are dimension tables whose keys are  $B$  and  $C$ , respectively. Typically, in a Data Warehouse query filters are specified on Dimensional Attributes. Hence each Map task has enough memory to hold the hash map of several small Dimensional tables. As Fact table rows are being read into the Map task, they can be hash joined with all the dimensional tables that the Map task has read into memory.

**Simple  $N$ -Way Joins.** A join on  $R(A, B)$ ,  $S(B, C)$ , and  $T(B, D)$  can be achieved in one MR job provided the rows for a key for all small tables can be buffered in memory. Suppose  $R$  is a large table and  $S$  and  $T$  are relatively smaller tables. Then it is typically the case that for any given key value  $B$ , the number of rows in  $S$  or  $T$  will fit in a Task's memory. Then, by giving the large table the largest tag, it is easy to generalize the Sort-Merge join to an  $N$ -way join where the joining expressions are the same. In a Reducer for a key value of  $B$ , the reducer will first receive the  $S$  rows, then the  $T$  rows, and finally the  $R$  rows. Since the assumption is that there aren't a large number of  $S$  and  $T$  rows, the reducer can cache them. As it receives  $R$  rows, it can do a cross product with the cached  $S$  and  $T$  rows and output the result of join.

In addition to the above strategies for performing joins using the MapReduce paradigm, algorithms have been proposed for other types joins (e.g., the general multi-way natural join with special cases of chain-join or star-join in data warehouses have been shown to be handled as a single MR job).<sup>14</sup> Similarly, algorithms have been proposed to deal with skew in the join attributes (e.g., in a sales fact table, certain days may have a disproportionate number of transactions). For joins on attributes with skew, a modified algorithm would let the *Partitioner* assign unique values to the

<sup>14</sup>See Afrati and Ullman (2010).

data having a large number of entries and let them be handled by Reduce tasks, whereas the rest of the values may undergo hash partitioning as usual.

This discussion should provide the reader with a good sense of the many possibilities of implementing Join strategies on top of MapReduce. There are other factors affecting performance, such as row versus columnar storage and pushing predicates down to storage handlers. These are beyond our scope of discussion here. Interested readers will find ongoing research publications in this area that are similar to Afrati and Ullman (2010).

The purpose of this section is to highlight two major developments that have impacted the big data community by providing high-level interfaces on top of the core technology of Hadoop and MapReduce. We will give a brief overview of the language Pig Latin and the system Hive.

**Apache Pig.** Pig<sup>15</sup> was a system that was designed at Yahoo Research to bridge the gap between declarative-style interfaces such as SQL, which we studied in the context of the relational model, and the more rigid low-level procedural-style programming style required by MapReduce that we described in Section 25.2.2. Whereas it is possible to express very complex analysis in MR, the user must express programs as a one-input, two-stage (map and reduce) process. Furthermore, MR provides no methods for describing a complex data flow that applies a sequence of transformations on the input. There is no standard way to do common data transformation operations like Projections, Filtering, Grouping, and Joining. We saw all these operations being expressed declaratively in SQL in Chapters 7 and 8. However, there is a community of users and programmers that thinks more procedurally. So the developers of Pig invented the language Pig Latin to fill in the “sweet spot” between SQL and MR. We show an example of a simple Group By query expressed in Pig Latin in Olston et al. (2008):

There is a table of urls: (url,category.pagerank).

We wish to find, for categories having a large number of URLs, the average pagerank of the high-pagerank URLs in that category. This requires a grouping of URLs by category. The SQL query that expresses this requirement may look like:

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10**6
```

The same query in Pig Latin is written as:

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)> 10**6;
output = FOREACH big_groups GENERATE
    category, AVG(good_urls.pagerank);
```

---

<sup>15</sup>See Olston et al. (2008).



As shown by this example, a Pigscript written using the scripting language Pig Latin is a sequence of data transformation steps. On each step, a basic transformation like Filter, Group By, or Projection is expressed. The script resembles a query plan for the SQL query similar to the plans we discussed in Chapter 19. The language supports operating on nested data structures like JSON (Java Script Object Notation) and XML. It has an extensive and extendible function library, and also an ability to bind schema to data very late or not at all.

Pig was designed to solve problems such as ad hoc analyses of Web logs and clickstreams. The logs and clickstreams typically require custom processing at row level as well as at an aggregate level. Pig accommodates user-defined functions (UDFs) extensively. It also supports a nested data model with the following four types:

**Atoms:** Simple atomic values such as a number or a string

**Tuples:** A sequence of fields, each of which can be of any permissible type

**Bag:** A collection of tuples with possible duplicates

**Map:** A collection of data items where each item has a key that allows direct access to it

Olston et al. (2008) demonstrates interesting applications on logs using Pig. An example is analysis of activity logs for a search engine over any time period (day, week, month, etc.) to calculate frequency of search terms by a user's geographic location. Here the functions needed include mapping IP addresses to geo-locations and using *n*-gram extraction. Another application involves co-grouping search queries of one period with those of another period in the past based on search terms.

Pig was architected so that it could run on different execution environments. In implementing Pig, Pig Latin was compiled into physical plans that were translated into a series of MR jobs and run in Hadoop. Pig has been a useful tool for enhancing programmers' productivity in the Hadoop environment.

### 25.4.3 Apache Hive

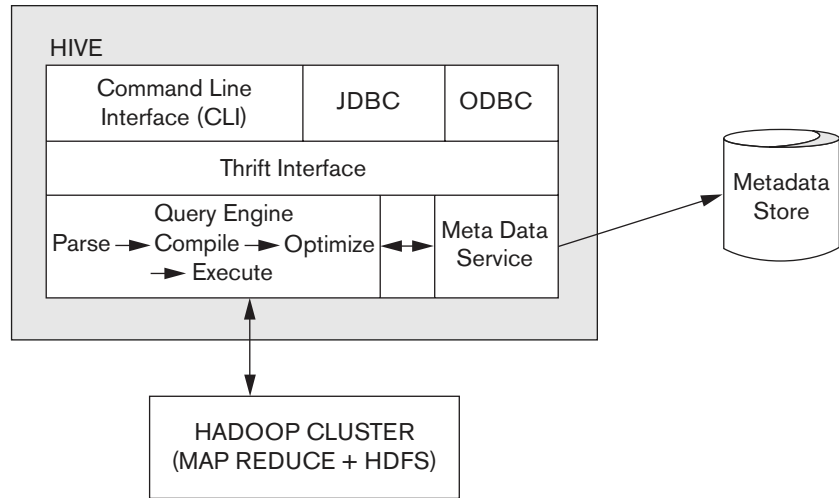
Hive was developed at Facebook<sup>16</sup> with a similar intent—to provide a higher level interface to Hadoop using SQL-like queries and to support the processing of aggregate analytical queries that are typical in data warehouses (see Chapter 29). Hive remains a primary interface for accessing data in Hadoop at Facebook; it has been adopted widely in the open source community and is undergoing continuous improvements. Hive went beyond Pig Latin in that it provided not only a high-level language interface to Hadoop, but a layer that makes Hadoop look like a DBMS with DDL, metadata repository, JDBC/ODBC access, and an SQL compiler. The architecture and components of Hive are shown in Figure 25.2.

Figure 25.2 shows Apache Thrift as interface in Hive. Apache Thrift defines an Interface Definition Language (IDL) and Communication Protocol used to develop

---

<sup>16</sup>See Thusoo et al. (2010).



**Figure 25.2**

Hive system architecture and components.

remote services. It comes with a runtime and code generation engine that can be used to develop remote services in many languages, including Java, C++, Python, and Ruby. Apache Thrift supports JSON-based and binary protocols; it supports http, socket, and file transports.

The Hive query language HiveQL includes a subset of SQL that includes all types of joins, Group By operations, as well as useful functions related to primitive and complex data types. We comment below on some of the highlights of the Hive system.

### Interfacing with HDFS:

- Tables in Hive are linked to directories in HDFS. Users can define partitions within tables. For example, a Web log table can be partitioned by day and within day by the hour. Each partition level introduces a level of directories in HDFS. A table may also be stored as *bucketed* on a set of columns. This means that the stored data is physically partitioned by the column(s). For example, within an hour directory, the data may be *bucketed* by Userid; this means that each hour's data is stored in a set of files, each file represents a bucket of Users, and the bucket is based on the hashing of the Userid column. Users can specify how many buckets the data should be divided into.
- The SerDe (Serialization/Deserialization) plugin architecture lets users specify how data in native file formats is exposed as rows to Hive SQL operators. Hive comes with a rich set of SerDe functions and supported File formats (e.g., CSV, JSON, SequenceFile); columnar formats (e.g., RCFile, ORCFile, Parquet); and support for Avro—another data serialization system. The different *StorageHandlers* expand on the SerDe mechanism to allow pluggable behavior for how data is read/written and the ability to push *predicates* down to the Storage Handler for early evaluation. For

example, the *JDBC StorageHandler* allows a Hive user to define a table that is in fact stored in some relational DBMS and accessed using the JDBC protocol (see Chapter 10) during query execution.

**Support of SQL and Optimizations in Hive:** Hive incorporated the concepts of Logical and Physical Optimizations similar to those used in optimization of SQL queries, which we discussed in Chapters 18 and 19. Early on, there was support for logical optimizations such as pruning unneeded columns and pushing selection predicates down into the query tree. Physical optimizations of converting sort-merge joins to Map-side joins based on user hints and data file sizes have also been incorporated. Hive started with support for a subset of SQL-92 that included SELECT, JOIN, GROUP BY, and filters based on conditions in the WHERE clause. Hive users can express complex SQL commands in Hive. Early in its development, Hive was able to run the 22 TPCB benchmark queries (Transaction Processing Performance Council benchmark for decision support), although with considerable manual rewriting.

Significant strides have been made in language support and in optimizer and run-time techniques. Here is a sampling of those improvements:

- Hive SQL has added many analytic features of SQL, such as subquery predicates, Common Table expressions (this is the WITH clause in SQL that allows users to name common subquery blocks and reference them multiple times in the query; these expressions can be considered query-level views), aggregates over a certain window within the data, Rollups (which refer to higher aggregation levels), and Grouping sets (this capability allows you to express multiple levels of aggregation in one Group By level). Consider, for example, Group By Grouping Sets ((year, month), (dayofweek)); this expresses aggregates both at the (Year, Month) level and also by DayOfWeek. A full set of SQL data types, including varchars, numeric types, and dates, is now supported. Hive also supports the common Change Data Capture ETL flow via Insert and Update statements. In a Data Warehouse, the process of delivering slowly changing Dimensions (e.g., customers in a Retail Data Warehouse) requires a complex dataflow of identifying new and updated records in that Dimension. This is called the Change Data Capture (CDC) process. By adding Insert and Update statements in Hive, it is possible to model and execute CDC processes in Hive SQL.
- Hive now has a greatly expanded set of DDLs for expressing grants and privileges in terms of discretionary access control (see Section 30.2).
- Several standard database optimizations have been incorporated, including Partition pruning, Join reordering, Index rewrite, and Reducing the number of MR jobs. Very large tables, like Fact tables in Data Warehouses, are typically partitioned. Time is probably the most common attribute used for partitioning. With HDFS being used as the storage layer, users tend to retain data for long time periods. But a typical Warehouse will only include the most current time periods (e.g., the last quarter or current year). The time periods are specified as filters in the Query. Partition Pruning is the technique of extracting relevant predicates from the Query filters and translating them to a list of

Table partitions that need to be read. Obviously, this has a huge impact on performance and cluster utilization: Instead of scanning all partitions retained for the last  $N$  years, only the partitions from the last few weeks/months are scanned. Work in progress includes collecting column- and table-level statistics and generating plans based on a cost model that uses these statistics (similar to what we considered for RDBMSs in Chapter 19).

- Hive now supports Tez as a runtime environment that has significant advantages over MR, including that there is no need to write to disk between jobs; and there is no restriction on one-input, two-stage processes. There is also active work to support Hive on Spark, a new technology that we briefly mention in Section 25.6.

#### 25.4.4 Advantages of the Hadoop/MapReduce Technology

Hadoop version 1 was optimized for batch processing on very large datasets. Various factors contribute to its success:

1. The disk seek rate is a limiting factor when we deal with petabyte-level workloads. Seek is limited by the disk mechanical structure, whereas the transfer speed is an electronic feature and increasing steadily. (See Section 16.2 for a discussion of disk drives.) The MapReduce model of scanning datasets in parallel alleviates this situation. For instance, scanning a 100-TB dataset sequentially using 1 machine at a rate of 50 Mbps will take about 24 days to complete. On the other hand, scanning the same data using 1,000 machines in parallel will just take 35 minutes. Hadoop recommends very large block sizes, 64 MB or higher. So when scanning datasets, the percentage of time spent on disk seeks is negligible. Unlimited disk seek rates combined with processing large datasets in chunks and in parallel is what drives the scalability and speed of the MapReduce model.
2. The MapReduce model allows handling of semistructured data and key-value datasets more easily compared to traditional RDBMSs, which require a predefined schema. Files such as very large logfiles present a particular problem in RDBMSs because they need to be parsed in multiple ways before they can be analyzed.
3. The MapReduce model has linear scalability in that resources can be added to improve job latency and throughput in a linear fashion. The failure model is simple, and individual failed jobs can be rerun without a major impact on the whole job.

### 25.5 Hadoop v2 alias YARN

In previous sections, we discussed Hadoop development in detail. Our discussion included the core concepts of the MapReduce paradigm for programming and the HDFS underlying storage infrastructure. We also discussed high-level interfaces like Pig and Hive that are making it possible to do SQL-like, high level data processing on top of the Hadoop framework. Now we turn our attention to subsequent developments, which are broadly called Hadoop v2 or MRv2 or YARN (Yet Another

Resource Negotiator). First, we point out the shortcomings of the Hadoop v1 platform and the rationale behind YARN.

### 25.5.1 Rationale behind YARN

Despite the success of Hadoop v1, user experience with Hadoop v1 in enterprise applications highlighted some shortcomings and suggested that an upgrade of Hadoop v1 might be necessary:

- As cluster sizes and the number of users grew, the JobTracker became a bottleneck. It was always known to be the Single Point of Failure.
- With a static allocation of resources to map and reduce functions, utilization of the cluster of nodes was less than desirable
- HDFS was regarded as a single storage system for data in the enterprise. Users wanted to run different types of applications that would not easily fit into the MR model. Users tended to get around this limitation by running Map-only Jobs, but this only compounded scheduling and utilization issues.
- On large clusters, it became problematic to keep up with new open source versions of Hadoop, which were released every few months.

The above reasons explain the rationale for developing version 2 of Hadoop. Some of the points mentioned in the previous list warrant a more detailed discussion, which we provide next.

**Multitenancy:** Multitenancy refers to accommodating multiple tenants/users concurrently so that they can share resources. As the cluster sizes grew and the number of users increased, several communities of users shared the Hadoop cluster. At Yahoo, the original solution to this problem was **Hadoop on Demand**, which was based on the Torque resource manager and Maui scheduler. Users could set up a separate cluster for each Job or set of Jobs. This had several advantages:

- Each cluster could run its own version of Hadoop.
- JobTracker failures were isolated to a single cluster.
- Each user/organization could make independent decisions on the size and configuration of its cluster depending on expected workloads.

But Yahoo abandoned Hadoop on Demand for the following reasons:

- Resource allocation was not based on data locality. So most reads and writes from HDFS were remote accesses, which negated one of the key benefits of the MR model of mostly local data accesses.
- The allocation of a cluster was static. This meant large parts of a cluster were mostly idle:
  - Within an MR job, the reduce slots were not usable during the Map phase and the map slots were not usable during the Reduce phase. When using higher level languages like Pig and Hive, each script or query spawned multiple Jobs. Since cluster allocation was static, the maximum nodes needed in any Job had to be acquired upfront.

- Even with the use of Fair or Capacity scheduling (see our discussion in Section 25.4.2), dividing the cluster into fixed map and reduce slots meant the cluster was underutilized.
- The latency involved in acquiring a cluster was high—a cluster would be granted only when enough nodes were available. Users started extending the lifetime of clusters and holding the clusters longer than they needed. This affected cluster utilization negatively.

**JobTracker Scalability.** As the cluster sizes increased beyond 4,000 nodes, issues with memory management and locking made it difficult to enhance JobTracker to handle the workload. Multiple options were considered, such as holding data about Jobs in memory, limiting the number of tasks per Job, limiting the number of Jobs submitted per user, and limiting the number of concurrently running jobs. None of these seemed to fully satisfy all users; JobTracker often ran out of memory.

A related issue concerned completed Jobs. Completed jobs were held in JobTracker and took up memory. Many schemes attempted to reduce the number and memory footprint of completed Jobs. Eventually, a viable solution was to offload this function to a separate Job History daemon.

As the number of TaskTrackers grew, the latencies for heartbeats (signals from TaskTracker to JobTracker) were almost 200 ms. This meant that heartbeat intervals for TaskTrackers could be 40 seconds or more when there were more than 200 task trackers in the cluster. Efforts were made to fix this but were eventually abandoned.

**JobTracker: Single Point of Failure.** The recovery model of Hadoop v1 was very weak. A failure of JobTracker would bring down the entire cluster. In this event, the state of running Jobs was lost, and all jobs would have to be resubmitted and JobTracker restarted. Efforts to make the information about completed jobs persist did not succeed. A related issue was to deploy new versions of the software. This required scheduling a cluster downtime, which resulted in backlogs of jobs and a subsequent strain on JobTracker upon restart.

**Misuse of the MapReduce Programming Model.** MR runtime was not a great fit for iterative processing; this was particularly true for machine learning algorithms in analytical workloads. Each iteration is treated as an MR job. Graph algorithms are better expressed using a bulk synchronous parallel (BSP) model, which uses message passing as opposed to the Map and Reduce primitives. Users got around these impediments by inefficient alternatives such as implementing machine learning algorithms as long-running Map-only jobs. These types of jobs initially read data from HDFS and executed the first pass in parallel; but then exchanged data with each other outside the control of the framework. Also, the fault tolerance was lost. The JobTracker was not aware of how these jobs operated; this lack of awareness led to poor utilization and instability in the cluster.

**Resource Model Issues.** In Hadoop v1, a node is divided into a fixed number of Map and Reduce slots. This led to cluster underutilization because idle slots could

not be used. Jobs other than MR could not run easily on the nodes because the node capacity remained unpredictable.

The aforementioned issues illustrate why Hadoop v1 needed upgrading. Although attempts were made to fix in Hadoop v1 many of the issues listed above, it became clear that a redesign was needed. The goals of the new design were set as follows:

- To carry forward the scalability and locality awareness of Hadoop v1.
- To have multitenancy and high cluster utilization.
- To have no single point of failure and to be highly available.
- To support more than just MapReduce jobs. The cluster resources should not be modeled as static map and reduce slots.
- To be backward compatible, so existing jobs should run as they are and possibly without any recompilation.

The outcome of these was YARN or Hadoop v2, which we discuss in the next section.

## 25.5.2 YARN Architecture

**Overview.** Having provided the motivation behind upgrading Hadoop v1, we now discuss the detailed architecture of the next generation of Hadoop, which is popularly known as MRv2, MapReduce 2.0, Hadoop v2, or YARN.<sup>17</sup> The central idea of YARN is the separation of cluster Resource Management from Jobs management. Additionally, YARN introduces the notion of an *ApplicationMaster*, which is now responsible for managing work (task data flows, task lifecycles, task failover, etc.). MapReduce is now available as a service/application provided by the *MapReduce ApplicationMaster*. The implications of these two decisions are far-reaching and are central to the notion of a data service operating system. Figure 25.3 shows a high-level schematic diagram of Hadoop v1 and Hadoop v2 side by side.

The *ResourceManager* and the per worker node *NodeManager* together form the platform on which any Application can be hosted on YARN. The *ResourceManager* manages the cluster, doling out Resources based on a pluggable scheduling policy (such as a fairness policy or optimizing cluster utilization policy). It is also responsible for the lifecycle of nodes in the cluster in that it will track when nodes go down, when nodes become unreachable, or when new nodes join. Node failures are reported to the *ApplicationMasters* that had containers on the failed node. New nodes become available for use by *ApplicationMasters*.

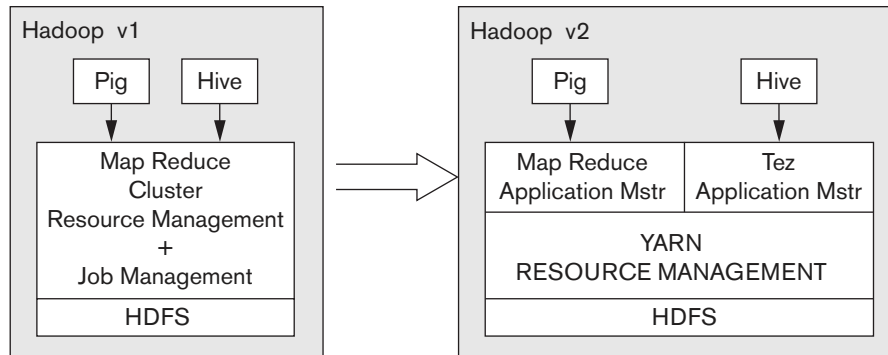
*ApplicationMasters* send *ResourceRequests* to the *ResourceManager* which then responds with cluster Container leases. A **Container** is a lease by the *ResourceManager* to the *ApplicationManager* to use certain amount of resources on a node

---

<sup>17</sup>See the Apache website: <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html> for up-to-date documentation on YARN.

**Figure 25.3**

The Hadoop v1 vs. Hadoop v2 schematic.



of the cluster. The *ApplicationMaster* presents a *Container Launch Context* to the *NodeManager* for the node that this lease references. The *Launch Context*, in addition to containing the lease, also specifies how to run the process for the task and how to get any resources like jars, libs for the process, environment variables, and security tokens. A node has a certain processing power in terms of number of cores, memory, network bandwidth, etc. Currently, YARN only considers memory. Based on its processing power, a node can be divided into an interchangeable set of containers. Once an *ApplicationMaster* receives a container lease, it is free to schedule work on it as it pleases. *ApplicationMasters*, based on their workload, can continuously change their Resource requirements. The *ResourceManager* bases its scheduling decisions purely on these requests, on the state of the cluster, and on the cluster's scheduling policy. It is not aware of the actual tasks being carried out on the nodes. The responsibility of managing and analyzing the actual work is left to *ApplicationMasters*.

The *NodeManager* is responsible for managing Containers on their nodes. Containers are responsible for reporting on the node health. They also handle the procedure for nodes joining the cluster. Containers provide the *Container Launch* service to *ApplicationMasters*. Other services available include a Local cache, which could be User level, Application level, or Container level. Containers also can be configured to provide other services to Tasks running on them. For example, for MR tasks, the shuffle is now provided as a Node-level service.

The *ApplicationMaster* is now responsible for running jobs on the cluster. Based on their job(s) the clusters negotiate for Resources with the *ResourceManager*. The *ApplicationMaster* itself runs on the cluster; at startup time a client submits an Application to the *ResourceManager*, which then allocates a container for the *ApplicationMaster* and launches it in that container. In the case of MR, the *ApplicationMaster* takes over most of the tasks of the *JobTracker*: it launches Map and Reduce tasks, makes decisions on their placement, manages failover of tasks, maintains counters similar to Job state counters, and provides a monitoring interface for running Jobs. The management and interface for completed jobs has been moved to a separate Job History Server.



The following advantages accrue from the separation of Resource Management from Application Management in the YARN architecture:

- A rich diversity of Data Services is available to utilize the cluster. Each of these can expose its own programming model.
- Application Masters are free to negotiate resources in patterns that are optimized for their work: for example, machine learning Apps may hold Containers for long durations.
- The Resource and Container model allows nodes to be utilized in a dynamic manner, which increases the overall utilization of the cluster.
- The ResourceManager does only one thing—manage resources; hence it is highly scalable to tens of thousands of nodes.
- With ApplicationMasters managing Jobs, it is possible to have multiple versions of an Application running on the cluster. There is no need for a global cluster update, which would require that all Jobs be stopped.

Failure of an ApplicationMaster affects only Jobs managed by it. The ResourceManager provides some degree of management of ApplicationMasters. Let us briefly consider each of the components of the YARN environment.

**Resource Manager (RM).** The Resource Manager is only concerned with allocating resources to Applications, and not with optimizing the processing within Applications. The policy of resource allocation is pluggable. Application Masters are supposed to request resources that would optimize their workload.

The Resource Manager exposes the following interfaces:

1. An API for clients to start ApplicationMasters
2. A protocol for ApplicationMasters to negotiate for cluster resources
3. A protocol for NodeManagers to report on node resources and be managed by the Resource Manager

The scheduler in the ResourceManager matches the Resource Requirements submitted by Applications against the global state of the cluster resources. The allocation is based on the policies of the pluggable Scheduler (such as capacity or fairness). Resources are requested by ApplicationMasters as **Resource Requests**. A Resource Request specifies:

- The number of containers needed
- The physical resources (CPU, memory) needed per container
- The locality preferences (physical node, rack) of the containers
- The priority of the request for the Application

The scheduler satisfies these requests based on the state of the cluster as reported by the NodeManager heartbeats. The locality and priority guides the scheduler toward alternatives: for example, if a requested node is busy, the next best alternative is another node on the same rack.



The scheduler also has the ability to request resources back from an Application if needed and can even take back the resources forcibly. Applications, in returning a container, can migrate the work to another container, or checkpoint the state and restore it on another container. It is important to point out what the Resource manager is *not* responsible for: handling the execution of tasks within an application, providing any status information about applications, providing history of finished jobs, and providing any recovery for failed tasks.

**ApplicationMaster (AM).** The ApplicationMaster is responsible for coordinating the execution of an Application on the cluster. An Application can be a set of processes like an MR Job, or it can be a long-running service like a Hadoop on demand (HOD) cluster serving multiple MR jobs. This is left to the Application Writer.

The ApplicationMaster will periodically notify the ResourceManager of its current Resource Requirements through a heartbeat mechanism. Resources are handed to the ApplicationMaster as Container leases. Resources used by an Application are dynamic: they are based on the progress of the application and the state of the cluster. Consider an example: the MR ApplicationMaster running an MR job will ask for a container on each of the  $m$  nodes where an InputSplit resides. If it gets a container on one of the nodes, the ApplicationMaster will either remove the request for containers on the rest of the  $m-1$  nodes or at least reduce their priority. On the other hand, if the map task fails, it is AM that tracks this failure and requests containers on other nodes that have a replica of the same InputSplit.

**NodeManager.** A NodeManager runs on every worker node of the cluster. It manages Containers and provides pluggable services for Containers. Based on a detailed *Container Launch Context* specification, a NodeManager can launch a process on its node with the environment and local directories set up. It also monitors to make sure the resource utilization does not exceed specifications. It also periodically reports on the state of the Containers and the node health. A NodeManager provides local services to all Containers running on it. The *Log Aggregation* service is used to upload each task's standard output and standard error (stdout and stderr) to HDFS. A NodeManager may be configured to run a set of pluggable *auxillary services*. For example, the MR Shuffle is provided as a NodeManager service. A Container running a Map task produces the Map output and writes to local disk. The output is made available to Reducers of the Job via the Shuffle service running on the Node.

**Fault tolerance and availability.** The RM remains the single point of failure in YARN. On restart, the RM can recover its state from a persistent store. It kills all containers in the cluster and restarts each ApplicationMaster. There is currently a push to provide an active/passive mode for RMs. The failure of an ApplicationMaster is not a catastrophic event; it only affects one Application. It is responsible for recovering the state of its Application. For example, the MR ApplicationMaster will recover its completed task and rerun any running tasks.

Failure of a Container because of issues with the Node or because of Application code is tracked by the framework and reported to the ApplicationMaster. It is the responsibility of the ApplicationMaster to recover from the failure.

### 25.5.3 Other Frameworks on YARN

The YARN architecture described above has made it possible for other application frameworks to be developed as well as other programming models to be supported that can provide additional services on the shared Hadoop cluster. Here we list some of the Frameworks that have become available in YARN at the time this text was written.

**Apache Tez.** Tez is an extensible framework being developed at Hortonworks for building high-performance applications in YARN; these applications will handle large datasets up to petabytes. Tez allows users to express their workflow as a directed acyclic graph (DAG) of tasks. Jobs are modeled as DAGs, where Vertices are tasks or operations and Edges represent interoperation dependencies or flows of data. Tez supports the standard dataflow patterns like pipeline, scatter-gather, and broadcast. Users can specify the concurrency in a DAG, as well as the failover characteristics, such as whether to store task output in persistent storage or to recompute it. The DAG can be changed at runtime based on job and cluster state. The DAG model is a more natural fit (than executing as one or more MapReduce jobs) for Pig scripts and SQL physical plans. Both Hive and Pig now provide a mode in which they run on Tez. Both have benefitted in terms of simpler plans and significant performance improvements. An often cited performance optimization is the Map-Reduce-Reduce pattern; an SQL query that has a Join followed by a Group-By normally is translated to two MR jobs: one for the Join and one for the Group-By. In the first MR stage, the output of the join will be written to HDFS and read back in the Map phase of the second MR for the Group-By Job. In Tez, this extra write and read to/from HDFS can be avoided by having the Join Vertex of the DAG stream resulting rows to the Group-By Vertex.

**Apache Giraph.** Apache Giraph is the open source implementation of Google's Pregel system,<sup>18</sup> which was a large-scale graph processing system used to calculate Page-Rank. (See Section 27.7.3 for a definition of Page-Rank.) Pregel was based on the bulk synchronous processing (BSP) model of computation.<sup>19</sup> Giraph added several features to Pregel, including sharded aggregators (sharding, as defined in Chapter 24, refers to a form of partitioning) and edge-oriented input. The Hadoop v1 version of Giraph ran as MR jobs, which was not a very good fit. It did this by running long-running Map-only Jobs. On YARN, the Giraph implementation exposes an iterative processing model. Giraph is currently used at Facebook to analyze the social network users' graph, which has users as nodes and their connections as edges; the current number of users is approximately 1.3 billion.

**Hoya: HBase on YARN.** The Hortonworks Hoya (HBase on YARN) project provides for elastic HBase clusters running on YARN with the goal of more flexibility and improved utilization of the cluster. We discussed HBase in Section 24.5 as a

---

<sup>18</sup>Pregel is described in Malewicz et al. (2010).

<sup>19</sup>BSP is a model for designing parallel algorithms and was originally proposed by Valiant (1990).

distributed, open source, nonrelational database that manages tables with billions of rows and millions of columns. HBase is patterned after BigTable from Google<sup>20</sup> but is implemented using Hadoop and HDFS. Hoya is being developed to address the need for creating on-demand clusters of HBase, with possibly different versions of HBase running on the same cluster. Each of the HBase instances can be individually configured. The Hoya ApplicationMaster launches the HBase Master locally. The Hoya AM also asks the YARN RM for a set of containers to launch HBase RegionServers on the cluster. HBase RegionServers are the worker processes of Hbase; each ColumnFamily (which is like a set of Columns in a relational table) is distributed across a set of RegionServers. This can be used to start one or more HBase instances on the cluster, on demand. The clusters are elastic and can grow or shrink based on demand.

The above three examples of the applications developed on YARN should give the reader a sense of the possibilities that have been opened up by the decoupling of Resource Management from Application Management in the overall Hadoop/MapReduce architecture by YARN.

## 25.6 General Discussion

So far, we have discussed the big data technology development that has occurred roughly in the 2004–2014 time frame, and we have emphasized Hadoop v1 and YARN (also referred to as Hadoop v2 or MRv2). In this section, we must first state the following disclaimer: there are a number of ongoing projects under Apache open source banner as well as in companies devoted to developing products in this area (e.g., Hortonworks, Cloudera, MapR) as well as many private startup companies. Similarly, the Amplab at University of California and other academic institutions are contributing heavily to developing technology that we have not been able to cover in detail. There is also a series of issues associated with the cloud concept, with running MapReduce in the cloud environment, and with data warehousing in the cloud that we have not discussed. Given this background, we now cover a few general topics that are worth mentioning in the context of the elaborate descriptions we presented so far in this chapter. We present issues related to the tussle between the traditional approach to high performance applications in parallel RDBMS implementations vis-à-vis Hadoop- and YARN-based technologies. Then we present a few points related to how big data and cloud technologies will be complementary in nature. We outline issues related to the locality of data and the optimization issues inherent in the storage clouds and the compute clouds. We also discuss YARN as a data services platform and the ongoing movement to harness big data for analytics. Finally, we present some current challenges facing the entire big data movement.

### 25.6.1 Hadoop/MapReduce vs. Parallel RDBMS

A team of data experts, including Abadi, DeWitt, Madden, and Stonebracker, have done a methodological study comparing a couple of parallel database systems with

---

<sup>20</sup>BigTable is described in Chang et al. (2006).

the open source version of Hadoop/MR (see, for example, Pavlo et al. (2009)). These experts measure the performance of these two approaches on the same benchmark using a 100-node cluster. They admit that the parallel database took longer to load and tune compared to MR, but the performance of parallel DBMSs was “strikingly better.” We list the areas the experts compared in the study and attempt to show the progress made in both DBMSs and Hadoop since then.

**Performance.** In their paper, Pavlo et al. concluded that parallel DBMSs were three to six times faster than MR. The paper lists many reasons why the DBMSs gave better performance. Among the reasons given are the following: (i) indexing with  $B^+$ -trees, which expedites selection and filtering; (ii) novel storage orientation (e.g., column-based storage has certain advantages); (iii) techniques that allow operations on compressed data directly; and (iv) parallel query optimization techniques common in parallel DBMSs.

Since the time of Pavlo et al.’s comparison, which involved Hadoop version 0.19, huge strides have been made in the MR runtime, the storage formats, and the planning capabilities for job scheduling and for optimizing complex data flows in the Hadoop ecosystem. *ORC* and *Parquet* file formats are sophisticated Columnar file formats that have the same aggressive compression techniques, the ability to push predicates to the storage layer, and the ability to answer aggregate queries without scanning data. We will briefly talk about the improvements in HDFS and MR; Apache Hive has made huge strides in both the runtime and Cost-based optimizations of complex SQLs. In their move to transform Hadoop from batch into real-time and interactive query mode, Hortonworks (2014) reports orders-of-magnitude gains in performance of queries on a TPC-DS (decision support )-style benchmark. Cloudera’s Impala product, as reported in Cloudera (2014), uses Parquet (the open source columnar data format) and is claimed to perform comparably to traditional RDBMSs.

**Upfront Cost advantage.** Hadoop has maintained its cost advantage. With few exceptions, Hadoop continues to be primarily an open source platform. YARN, Hive, and Spark are all developed as Apache projects and are available as freely downloadable packages.

**Handling Unstructured/Semistructured data.** MR reads data by applying the schema definition to it; doing so allows it to handle semistructured datasets like CSVs, JSON, and XML documents. The loading process is relatively inexpensive for the Hadoop/MR systems. However, the support for unstructured data is definitely on the rise in RDBMSs. PostgreSQL now supports key-value stores and json; most RDBMSs have a support for XML. On the other hand, one of the reasons for the performance gains on the Hadoop side has been the use of specialized data formats like ORC (Optimized Row Columnar) and Parquet (another open source columnar format). The latter may not remain a strongly differentiating feature among RDBMSs and Hadoop-based systems for too long because RDBMSs may also incorporate special data formats.

**Higher level language support.** SQL was a distinguishing feature that was in favor for RDBMSs for writing complex analytical queries. However, Hive has incorporated a large number of SQL features in HiveQL, including grouping and aggregation as well as nested subqueries and multiple functions that are useful in data warehouses, as we discussed previously. Hive 0.13 is able to execute about 50 queries from the TPC-DS benchmark without any manual rewriting. New machine learning-oriented function libraries are emerging (e.g., the function library at [madlib.net](http://madlib.net) supports traditional RDBMSs like PostgreSQL as well as the Pivotal distribution of Hadoop database (PHD)). Pivotal's HAWQ claims to be the latest and most powerful parallel SQL engine combining the advantages of SQL and Hadoop. Furthermore, the YARN plugin architecture that we discussed simplifies the process of extending the fabric with new components and new functions. Pig and Hive have extensibility with UDFs (user-defined functions). Several data services are now available on YARN, such as Revolution R and Apache Mahout for machine learning and Giraph for graph processing. Many traditional DBMSs now run on the YARN platform; for example, the Vortex analytic platform from Actian<sup>21</sup> and BigSQL 3.0 from IBM.<sup>22</sup>

**Fault tolerance.** Fault tolerance remains a decided advantage of MR-based systems. The panel of authors in Pavlo et al. (2009) also acknowledged that “MR does a superior job of minimizing the amount of work lost when a hardware failure occurs.” As pointed out by these authors, this capability comes at the cost of materializing intermediate files between Map and Reduce phases. But as Hadoop begins to handle very complex data flows (such as in Apache Tez) and as the need for latencies decreases, users can trade off performance for fault tolerance. For example, in Apache Spark one can configure an intermediate Resilient Distributed Dataset (RDD)<sup>23</sup> to be either materialized on disk or in memory, or even to be recomputed from its input.

As we can see from this discussion, even though MR started with a goal of supporting batch-oriented workloads, it could not keep up with traditional parallel RDBMSs in terms of interactive query workloads, as exemplified by Pavlo et al. (2009). However, the two camps have moved much closer to each other in capabilities. Market forces, such as the need for venture capital for new startups, require an SQL engine for new applications that largely deal with very large semistructured datasets; and the research community's interest and involvement have brought about substantial improvements in Hadoop's capability to handle traditional analytical workloads. But there is still significant catching up to be done in all the areas pointed out in Pavlo et al. (2009): runtime, planning and optimization, and analytic feature-sets.

<sup>21</sup>See <http://www.actian.com/about-us/blog/sql-hadoop-real-deal/> for a current description.

<sup>22</sup>See Presentation at [http://www.slideshare.net/Hadoop\\_Summit/w-325p230-azubirigrayatv4](http://www.slideshare.net/Hadoop_Summit/w-325p230-azubirigrayatv4) for a current description.

<sup>23</sup>See Zaharia et al. (2012).

### 25.6.2 Big Data in Cloud Computing

The cloud computing movement and the big data movement have been proceeding concurrently for more than a decade. It is not possible to address the details of cloud computing issues in the present context. However, we state some compelling reasons why big data technology is in some sense dependent on cloud technology not only for its further expansion, but for its continued existence.

- The cloud model affords a high degree of flexibility in terms of management of resources: “scaling out,” which refers to adding more nodes or resources; “scaling up,” which refers to adding more resources to a node in the system; or even downgrading are easily handled almost instantaneously.
- The resources are interchangeable; this fact, coupled with the design of distributed software, creates a good ecosystem where failure can be absorbed easily and where virtual computing instances can be left unperturbed. For the cost of a few hundred dollars, it is possible to perform data mining operations that involve complete scans of terabyte databases, and to crawl huge Web sites that contain millions of pages.
- It is not uncommon for big data projects to exhibit unpredictable or peak computing power and storage needs. These projects are faced with the challenge of providing for this peak demand on an as-needed and not necessarily continuous basis. At the same time, business stakeholders expect swift, inexpensive, and dependable products and project outcomes. To meet with these conflicting requirements, cloud services offer an ideal solution.
- A common situation in which cloud services and big data go hand-in-hand is as follows: Data is transferred to or collected in a cloud data storage system, like Amazon’s S3, for the purpose of collecting log files or exporting text-formatted data. Alternatively, database adapters can be utilized to access data from databases in the cloud. Data processing frameworks like Pig, Hive, and MapReduce, which we described above in Section 25.4, are used to analyze raw data (which may have originated in the cloud).
- Big data projects and startup companies benefit a great deal from using a cloud storage service. They can trade capital expenditure for operational expenditure; this is an excellent trade because it requires no capital outlay or risk. Cloud storage provides reliable and scalable storage solutions of a quality otherwise unachievable.
- Cloud services and resources are globally distributed. They ensure high availability and durability unattainable by most but the largest organizations.

**The Netflix Case for Marrying Cloud and Big Data.**<sup>24</sup> Netflix is a large organization characterized by a very profitable business model and an extremely inexpensive and reliable service for consumers. Netflix provides video streaming services to millions of customers today thanks to a highly efficient information

---

<sup>24</sup>Based on <http://techblog.netflix.com/2013/01/hadoop-platform-as-service-in-cloud.html>



system and data warehouse. Netflix uses Amazon S3 rather than HDFS as the data processing and analysis platform for several reasons. Netflix presently uses Amazon's Elastic MapReduce (EMR) distribution of Hadoop. Netflix cites the main reason for its choice as the following: S3 is designed for 99.999999999% durability and 99.99% availability of objects over a given year, and S3 can sustain concurrent loss of data in two facilities. S3 provides bucket versioning, which allows Netflix to recover inadvertently deleted data. The elasticity of S3 has allowed Netflix a practically unlimited storage capacity; this capacity has enabled Netflix to grow its storage from a few hundred terabytes to petabytes without any difficulty or prior planning. Using S3 as the data warehouse enables Netflix to run multiple Hadoop clusters that are fault-tolerant and can sustain excess load. Netflix executives claim that they have no concerns about data redistribution or loss during expansion or shrinking of the warehouse. Although Netflix's production and query clusters are long-running clusters in the cloud, they can be essentially treated as completely transient. If a cluster goes down, Netflix can simply substitute with another identically sized cluster, possibly in a different geographic zone, in a few minutes and not sustain any data loss.

### 25.6.3 Data Locality Issues and Resource Optimization for Big Data Applications in a Cloud

The increasing interest in cloud computing combined with the demands of big data technology means that data centers must be increasingly cost-effective and consumer-driven. Also, many cloud infrastructures are not intrinsically designed to handle the scale of data required for present-day data analytics. Cloud service providers are faced with daunting challenges in terms of resource management and capacity planning to provide for big data technology applications.

The network load of many big data applications, including Hadoop/MapReduce, is of special concern in a data center because large amounts of data can be generated during job execution. For instance, in a MapReduce job, each reduce task needs to read the output of all map tasks, and a sudden explosion of network traffic can significantly deteriorate cloud performance. Also, when data is located in one infrastructure (say, in a storage cloud like Amazon S3) and processed in a compute cloud (such as Amazon EC2), job performance suffers significant delays due to data loading.

Research projects have proposed<sup>25</sup> a self-configurable, locality-based data and virtual machine management framework based on the storage-compute model. This framework enables MapReduce jobs to access most of their data either locally or from close-by nodes, including all input, output, and intermediate data generated during map and reduce phases of the jobs. Such frameworks categorize jobs using a data-size sensitive classifier into four classes based on a data size-based footprint. Then they provision virtual MapReduce clusters in a locality-aware manner, which enables efficient pairing and allocation of MapReduce virtual machines (VMs) to reduce the network distance between storage and compute nodes for both map and reduce processing.

---

<sup>25</sup>See Palanisamy et al. (2011).

Recently, caching techniques have been shown to improve the performance of MapReduce jobs for various workloads.<sup>26</sup> The PACMan framework provides support for in-memory caching, and the MixApart system provides support for disk-based caching when the data is stored in an enterprise storage server within the same site. Caching techniques allow flexibility in that data is stored in a separate storage infrastructure that allows prefetching and caching of the most essential data. Recent work<sup>27</sup> has addressed the big data caching problem in the context of privacy-conscious scenarios, wherein data stored in encrypted form in a public cloud must be processed in a separate, secure enterprise site.

In addition to the data locality problem, one of the most challenging goals for cloud providers is to optimally provision virtual clusters for jobs while minimizing the overall consumption cost of the cloud data center.

An important focus of cloud resource optimization is to optimize globally across all jobs in the cloud as opposed to per-job resource optimizations. A good example of a globally optimized cloud- managed system is the recent Google BigQuery system,<sup>28</sup> which allows Google to run SQL-like queries against very large datasets with potentially billions of rows using an Excel-like interface. In the BigQuery service, customers only submit the queries to be processed on the large datasets, and the cloud system intelligently manages the resources for the SQL-like queries. Similarly, the Cura resource optimization model<sup>29</sup> proposed for MapReduce in a cloud achieves global resource optimization by minimizing the overall resource utilization in the cloud as opposed to per-job or per-customer resource optimization.

## 25.6.4 YARN as a Data Service Platform

The separation of resource management from application management has taken Hadoop to another level as a platform. Hadoop v1 was all about MapReduce. In Hadoop v2, MapReduce is one of the many application frameworks that can run on the cluster. As we discussed in Section 25.5, this has opened the door for many services (with their own programming models) to be provided on YARN. There is no need to translate all data processing techniques and algorithms into a set of MapReduce jobs. MapReduce is presently being used only for batch-oriented processing such as the ETL (extract, transform, load) process in data warehouses (see Chapter 29). The emerging trend is to see Hadoop as a **data lake**, where a significant portion of enterprise data resides and where processing happens. Traditionally, HDFS has been where an enterprise's historical data resides because HDFS can handle the scale of such data. Most new sources of data, which in today's search and social networking applications come from Web and machine logs, clickstream data, message data (as in Twitter) and sensor data, also is being stored largely in HDFS.

<sup>26</sup>See the PACMAN framework by Ananthanarayanan et al. (2012) and the MixApart system by Mihailescu et al. (2013).

<sup>27</sup>See Palanisamy et al. (2014a).

<sup>28</sup>For the Google BigQuery system, see <https://developers.google.com/bigquery/>

<sup>29</sup>Palanisamy et al. (2014b).



The Hadoop v1 model was the **federation** model: although HDFS was the storage layer for the enterprise, processing was a mixture of MapReduce and other engines. One alternative was to extract data from HDFS store to engines running outside the cluster in their own silos; such data was moved to graph engines, machine learning analytical applications, and so forth. The same machines as those used for the Hadoop cluster were being used for entirely different applications, such as stream processing outside of Hadoop. This scenario was far from ideal since physical resources had to be divvied up in a static manner and it was difficult to migrate and upgrade to new versions when multiple frameworks ran on the same machines. With YARN, the above issues are addressed. Traditional services are taking advantage of the YARN ResourceManager and are providing their service on the same Hadoop cluster where the data resides.

Whereas support for SQL in Hadoop was promised by multiple vendors, the actual support has been less than completely desirable. Some vendors required the HDFS data to be moved out to another database to run SQL; some required wrappers to read the HDFS data before an SQL query ran on it. A new trend among RDBMSs and traditional database systems considers a YARN cluster as a viable platform. One example is Actian's analytics platform, which provides SQL in Hadoop<sup>30</sup> and which is claimed to be a complete and robust implementation of SQL using the Actian Vectorwise columnar database (which runs as a YARN application). IBM's Big SQL 3.0<sup>31</sup> is a project that makes an existing IBM shared-nothing DBMS run on a YARN cluster.

Apache Storm is a distributed scalable streaming engine that allows users to process real-time data feeds. It is widely used by Twitter. Storm on YARN (<http://hortonworks.com/labs/storm/>) and SAS on YARN (<http://hortonworks.com/partner/sas/>) are applications that treat Storm (a distributed stream processing application) and SAS (statistical analysis software) as applications on the YARN platform. As we discussed previously, Giraph and HBase Hoya are ongoing efforts that are rapidly adopting YARN. A wide range of application systems uses the Hadoop cluster for storage; examples include services like streaming, machine learning/statistics, graph processing, OLAP, and key-value stores. These services go well beyond MapReduce. The goal/promise of YARN is for these services to coexist on the same cluster and take advantage of the locality of data in HDFS while YARN orchestrates their use of cluster resources.

### 25.6.5 Challenges Faced by Big Data Technologies

In a recent article,<sup>32</sup> several database experts voiced their concerns about the impending challenges faced by big data technologies when such technologies

---

<sup>30</sup>Current documentation is available at <http://www.actian.com/about-us/blog/sql-hadoop-real-deal/>

<sup>31</sup>Current information is available at: [http://www.slideshare.net/Hadoop\\_Summit/w-325p230-azubirigraytv4](http://www.slideshare.net/Hadoop_Summit/w-325p230-azubirigraytv4)

<sup>32</sup>See Jagadish et al. (2014).

are used primarily for analytics applications. These concerns include the following:

- **Heterogeneity of information:** Heterogeneity in terms of data types, data formats, data representation, and semantics is unavoidable when it comes to sources of data. One of the phases in the big data life cycle involves integration of this data. The cost of doing a clean job of integration to bring all data into a single structure is prohibitive for most applications, such as health-care, energy, transportation, urban planning, and environmental modeling. Most machine learning algorithms expect data to be fed into them in a uniform structure. The data provenance (which refers to the information about the origin and ownership of data) is typically not maintained in most analytics applications. Proper interpretation of data analysis results requires large amounts of metadata.
- **Privacy and confidentiality:** Regulations and laws regarding protection of confidential information are not always available and hence not applied strictly during big data analysis. Enforcement of HIPAA regulations in the healthcare environment is one of few instances where privacy and confidentiality are strictly enforced. Location-based applications (such as on smart phones and other GPS-equipped devices), logs of user transactions, and clickstreams that capture user behavior all reveal confidential information. User movement and buying patterns can be tracked to reveal personal identity. Because it is now possible to harness and analyze billions of users' records via the technologies described in this chapter, there is widespread concern about personal information being compromised (e.g., data about individuals could be leaked from social data networks that are in some way linked to other data networks). Data about customers, cardholders, and employees is held by organizations and thus is subject to breaches of confidentiality. Jagadish et al. (2014) voiced a need for stricter control over digital rights management of data similar to the control exercised in the music industry.
- **Need for visualization and better human interfaces:** Huge volumes of data are crunched by big data systems, and the results of analyses must be interpreted and understood by humans. Human preferences must be accounted for and data must be presented in a properly digestible form. Humans are experts at detecting patterns and have great intuition about data they are familiar with. Machines cannot match humans in this regard. It should be possible to bring together multiple human experts to share and interpret results of analysis and thereby increase understanding of those results. Multiple modes of visual exploration must be possible to make the best use of data and to properly interpret results that are out of range and thus are classified as outlier values.
- **Inconsistent and incomplete information:** This has been a perennial problem in data collection and management. Future big data systems will allow multiple sources to be handled by multiple coexisting applications, so problems due to missing data, erroneous data, and uncertain data will be compounded. The large volume and built-in redundancy of data in fault-tolerant

systems may compensate to some extent for the missing values, conflicting values, hidden relationships, and the like. There is an inherent uncertainty about data collected from regular users using normal devices when such data comes in multiple forms (e.g., images, rates of speed, direction of travel). There is still a lot to be learned about how to use crowdsourcing data to generate effective decision making.

The aforementioned issues are not new to information systems. However, the large volume and wide variety of information inherent in big data systems compounds these issues.

### 25.6.6 Moving Forward

YARN makes it feasible for enterprises to run and manage many services on one cluster. But building data solutions on Hadoop is still a daunting challenge. A solution may involve assembling ETL (extract, transform, load) processing, machine learning, graph processing, and/or report creation. Although these different functional engines all run on the same cluster, their programming models and metadata are not unified. Analytics application developers must try to integrate all these services into a coherent solution.

On current hardware, each node contains a significant amount of main memory and flash memory storage. The cluster thus becomes a vast resource of main memory and flash storage. Significant innovation has demonstrated the performance gains of **in-memory data engines**; for example, SAP HANA is an in-memory, columnar scale-out RDBMS that is gaining a wide following.<sup>33</sup>

The Spark platform from Databricks (<https://databricks.com/>), which is an offshoot of the Berkeley Data Analytics Stack from AMPLabs at Berkeley,<sup>34</sup> addresses both of the advances mentioned above—namely, the ability to house diverse applications in one cluster and the ability to use vast amounts of main memory for faster response. Matei Zaharia developed the Resilient Distributed Datasets (RDD) concept<sup>35</sup> as a part of his Ph.D. work at the University of California–Berkeley that gave rise to the Spark system. The concept is generic enough to be used across all Spark’s engines: Spark core (data flow), Spark-SQL, GraphX, (graph processing), MLlib (machine learning), and Spark-Streaming (stream processing). For example, it is possible to write a script in Spark that expresses a data flow that reads data from HDFS, reshapes the data using a Spark-SQL query, passes that information to an MLlib function for machine learning–type analysis, and then stores the result back in HDFS.<sup>36</sup>

<sup>33</sup>See <http://www.saphana.com/welcome> for a variety of documentation on SAP’s HANA system.

<sup>34</sup>See <https://amplab.cs.berkeley.edu/software/> for projects at Amplab from the University of California–Berkeley.

<sup>35</sup>The RDD concept was first proposed in Zaharia et al. (2012).

<sup>36</sup>See an example of the use of Spark at <https://databricks.com/blog/2014/03/26/spark-sql-manipulating-structured-data-using-spark-2.html>

RDDs are built on the capabilities of Scala language collections<sup>37</sup> that are able to re-create themselves from their input. RDDs can be configured based on how their data is distributed and how their data is represented: it can be always re-created from input, and it can be cached on disk or in memory. In-memory representations vary from serialized Java objects to highly optimized columnar formats that have all the advantages of columnar databases (e.g., speed, footprint, operating in serialized form).

The capabilities of a unified programming model and in-memory datasets will likely be incorporated into the Hadoop ecosystem. Spark is already available as a service in YARN (<http://spark.apache.org/docs/1.0.0/running-on-yarn.html>). Detailed discussion of Spark and related technologies in the Berkeley Data Analysis Stack is beyond our scope here. Agneeswaran (2014) discusses the potential of Spark and related products; interested readers should consult that source.

## 25.7 Summary

In this chapter, we discussed big data technologies. Reports from IBM, McKinsey, and TeraData scientist Bill Franks all predict a vibrant future for this technology, which will be at the center of future data analytics and machine learning applications and which is predicted to save businesses billions of dollars in the coming years.

We began our discussion by focusing on developments at Google with the Google file system and MapReduce (MR), a programming paradigm for distributed processing that is scalable to huge quantities of data reaching into the petabytes. After giving a historical development of the technology and mentioning the Hadoop ecosystem, which spans a large number of currently active Apache projects, we discussed the Hadoop distributed file system (HDFS) by outlining its architecture and its handling of file operations; we also touched on the scalability studies done on HDFS. We then gave details of the MapReduce runtime environment. We provided examples of how the MapReduce paradigm can be applied to a variety of contexts; we gave a detailed example of its application to optimizing various relational join algorithms. We then presented briefly the developments of Pig and Hive, the systems that provide an SQL-like interface with Pig Latin and HiveQL on top of the low-level MapReduce programming. We also mentioned the advantages of the joint Hadoop/MapReduce technology.

Hadoop/MapReduce is undergoing further development and is being repositioned as version 2, known as MRv2 or YARN; version 2 separates resource management from task/job management. We discussed the rationale behind YARN, its architecture, and other ongoing frameworks based on YARN, including Apache Tez, a workflow modeling environment; Apache Giraph, a large-scale graph processing system based on Pregel of Google; and Hoya, a Hortonworks rendering of HBase elastic clusters on YARN.

---

<sup>37</sup>See <http://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html> for more information on Scala Collections.

Finally, we presented a general discussion of some issues related to MapReduce/Hadoop technology. We briefly commented on the study done for this architecture vis-à-vis parallel DBMSs. There are circumstances where one is superior over the other, and claims about the superiority of parallel DBMSs for batch jobs are becoming less relevant due to architectural advancements in the form of YARN-related developments. We discussed the relationship between big data and cloud technologies and the work being done to address data locality issues in cloud storage for big data analytics. We stated that YARN is being considered as a generic data services platform, and we listed the challenges for this technology as outlined in a paper authored by a group of database experts. We concluded with a summary of ongoing projects in the field of big data.

## Review Questions

- 25.1. What is data analytics and what is its role in science and industry?
- 25.2. How will the big data movement support data analytics?
- 25.3. What are the important points made in the McKinsey Global Institute report of 2012?
- 25.4. How do you define big data?
- 25.5. What are the various types of analytics mentioned in the IBM (2014) book?
- 25.6. What are the four major characteristics of big data? Provide examples drawn from current practice of each characteristic.
- 25.7. What is meant by *veracity of data*?
- 25.8. Give the chronological history of the development of MapReduce/Hadoop technology.
- 25.9. Describe the execution workflow of the MapReduce programming environment.
- 25.10. Give some examples of MapReduce applications.
- 25.11. What are the core properties of a job in MapReduce?
- 25.12. What is the function of JobTracker?
- 25.13. What are the different releases of Hadoop?
- 25.14. Describe the architecture of Hadoop in your own words.
- 25.15. What is the function of the NameNode and secondary NameNode in HDFS?
- 25.16. What does the Journal in HDFS refer to? What data is kept in it?
- 25.17. Describe the heartbeat mechanism in HDFS.
- 25.18. How are copies of data (replicas) managed in HDFS?

- 25.19.** Shvachko (2012) reported on HDFS performance. What did he find? Can you list some of his results?
- 25.20.** What other projects are included in the open source Hadoop ecosystem?
- 25.21.** Describe the workings of the JobTracker and TaskTracker in MapReduce.
- 25.22.** Describe the overall flow of the job in MapReduce.
- 25.23.** What are the different ways in which MapReduce provides fault tolerance?
- 25.24.** What is the Shuffle procedure in MapReduce?
- 25.25.** Describe how the various job schedulers for MapReduce work.
- 25.26.** What are the different types of joins that can be optimized using MapReduce?
- 25.27.** Describe the MapReduce join procedures for Sort-Merge join, Partition Join, *N*-way Map-side join, and Simple *N*-way join.
- 25.28.** What is Apache Pig, and what is Pig Latin? Give an example of a query in Pig Latin.
- 25.29.** What are the main features of Apache Hive? What is its high-level query language?
- 25.30.** What is the SERDE architecture in Hive?
- 25.31.** List some of the optimizations in Hive and its support of SQL.
- 25.32.** Name some advantages of the MapReduce/Hadoop technology.
- 25.33.** Give the rationale in moving from Hadoop v1 to Hadoop v2 (YARN).
- 25.34.** Give an overview of the YARN architecture.
- 25.35.** How does Resource Manager work in YARN?
- 25.36.** What are Apache Tez, Apache Giraph, and Hoya?
- 25.37.** Compare parallel relational DBMSs and the MapReduce/Hadoop systems.
- 25.38.** In what way are big data and cloud technology complementary to one another?
- 25.39.** What are the data locality issues related to big data applications in cloud storage?
- 25.40.** What services can YARN offer beyond MapReduce?
- 25.41.** What are some of the challenges faced by big data technologies today?
- 25.42.** Discuss the concept of RDDs (resilient distributed datasets).
- 25.43.** Find out more about ongoing projects such as Spark, Mesos, Shark, and BlinkDB as they relate to the Berkeley Data Analysis Stack.

## Selected Bibliography

The technologies for big data discussed in this chapter have mostly sprung up in the last ten years or so. The origin of this wave is traced back to the seminal papers from Google, including the Google file system (Ghemawat, Gobioff, & Leung, 2003) and the MapReduce programming paradigm (Dean & Ghemawat, 2004). The Nutch system with follow-on work at Yahoo is a precursor of the Hadoop technology and continues as an Apache open source project ([nutch.apache.org](http://nutch.apache.org)). The BigTable system from Google (Fay Chang et al., 2006) describes a distributed scalable storage system for managing structured data in the petabytes range over thousands of commodity servers.

It is not possible to name a specific single publication as “the” Hadoop paper. Many studies related to MapReduce and Hadoop have been published in the past decade. We will list only a few landmark developments here. Schvachko (2012) outlines the limitations of the HDFS file system. Afrati and Ullman (2010) is a good example of using MapReduce programming in various contexts and applications; they demonstrate how to optimize relational join operations in MapReduce. Olston et al. (2008) describe the Pig system and introduce Pig Latin as a high-level programming language. Thusoo et al. (2010) describe Hive as a petabyte- scale data warehouse on top of Hadoop. A system for large-scale graph processing called Pregel at Google is described in Malewicz et al. (2010). It uses the bulk synchronous parallel (BSP) model of parallel computation originally proposed by Valiant (1990). In Pavlo et al. (2009), a number of database technology experts compared two parallel RDBMSs with Hadoop/MapReduce and showed how the parallel DBMS can actually perform better under certain conditions. The results of this study must not be considered definitive because of the significant performance improvements achieved in Hadoop v2 (YARN). The approach of resilient distributed datasets (RDDs) for in-memory cluster computing is at the heart of the Berkeley’s Spark system, developed by Zaharia et al. (2013). A recent paper by Jagadish et al. (2014) gives the collective opinion of a number of database experts about the challenges faced by the current big data technologies.

The definitive resource for Hadoop application developers is the book *Hadoop: The Definitive Guide*, by Tom White (2012), which is in its third edition. A book by YARN project founder Arun Murthy with Vavilapalli (2014) describes how YARN increases scalability and cluster utilization, enables new programming models and services, and extends applicability beyond batch applications and Java. Agneeswaran (2014) has written about going beyond Hadoop, and he describes the Berkeley Data Analysis Stack (BDAS) for real-time analytics and machine learning; the Stack includes Spark, Mesos, and Shark. He also describes Storm, a complex event-processing engine from Twitter widely used in industry today for real-time computing and analytics.

The Hadoop wiki is at [Hadoop.apache.org](http://Hadoop.apache.org). There are many open source, big data projects under Apache, such as Hive, Pig, Oozie, Sqoop, Storm, and HBase. Up-to-date information about these projects can be found in the documentation at the projects’ Apache Web sites and wikis. The companies Cloudera, MapR, and Hor-



tonworks include on their Web sites documentation about their own distributions of MapReduce/Hadoop-related technologies. The Berkeley Amplab (<https://amplab.cs.berkeley.edu/>) provides documentation about the Berkeley Data Analysis Stack (BDAS), including ongoing projects such as GraphX, MLbase, and BlinkDB.

There are some good references that outline the promise of big data technology and large scale data management. Bill Franks (2012) talks about how to leverage big data technologies for advanced analytics and provides insights that will help practitioners make better decisions. Schmarzo (2013) discusses how the big data analytics can empower businesses. Dietrich et al. (2014) describe how IBM has applied the power of big data analytics across the enterprise in applications worldwide. A book published by McKinsey Global Institute (2012) gives a strategic angle on big data technologies by focusing on productivity, competitiveness, and growth.

There has been a parallel development in the cloud technologies that we have not been able to discuss in detail in this chapter. We refer the reader to recent books on cloud computing. Erl et al. (2013) discusses models, architectures, and business practices and describes how this technology has matured in practice. Kavis (2014) presents the various service models, including software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS). Bahga and Madiseti (2013) offer a practical, hands-on introduction to cloud computing. They describe how to develop cloud applications on various cloud platforms, such as Amazon Web Service (AWS), Google Cloud, and Microsoft's Windows Azure.



This page intentionally left blank

part **11**

**Advanced Database Models,  
Systems, and Applications**

This page intentionally left blank

## Enhanced Data Models: Introduction to Active, Temporal, Spatial, Multimedia, and Deductive Databases

As the use of database systems has grown, users have demanded additional functionality from these software packages; increased functionality would make it easier to implement more advanced and complex user applications. Object-oriented databases and object-relational systems do provide features that allow users to extend their systems by specifying additional abstract data types for each application. However, it is useful to identify certain common features for some of these advanced applications and to create models that can represent them. Additionally, specialized storage structures and indexing methods can be implemented to improve the performance of these common features. Then the features can be implemented as abstract data types or class libraries and purchased separately from the basic DBMS software package. The term **data blade** has been used in Informix and **cartridge** in Oracle to refer to such optional submodules that can be included in a DBMS package. Users can utilize these features directly if they are suitable for their applications, without having to reinvent, reimplement, and reprogram such common features.

This chapter introduces database concepts for some of the common features that are needed by advanced applications and are being used widely. We will cover *active rules* that are used in active database applications, *temporal concepts* that are used in temporal database applications, and, briefly, some of the issues involving *spatial databases* and *multimedia databases*. We will also discuss *deductive databases*. It is important to note that each of these topics is very broad, and we give

only a brief introduction to each. In fact, each of these areas can serve as the sole topic of a complete book.

In Section 26.1, we introduce the topic of active databases, which provide additional functionality for specifying **active rules**. These rules can be automatically triggered by events that occur, such as database updates or certain times being reached, and can initiate certain actions that have been specified in the rule declaration to occur if certain conditions are met. Many commercial packages include some of the functionality provided by active databases in the form of **triggers**. Triggers are now part of the SQL-99 and later standards.

In Section 26.2, we introduce the concepts of **temporal databases**, which permit the database system to store a history of changes and allow users to query both current and past states of the database. Some temporal database models also allow users to store future expected information, such as planned schedules. It is important to note that many database applications are temporal, but they are often implemented without having much temporal support from the DBMS package—that is, the temporal concepts are implemented in the application programs that access the database. The ability to create and query temporal data has been added to the SQL standard in SQL:2011 and is available in the DB2 system, but we do not discuss it here. The interested reader is referred to the end-of-chapter bibliography.

Section 26.3 gives a brief overview of **spatial database** concepts. We discuss types of spatial data, different kinds of spatial analyses, operations on spatial data, types of spatial queries, spatial data indexing, spatial data mining, and applications of spatial databases. Most commercial and open source relational systems provide spatial support in their data types and query languages as well as providing indexing and efficient query processing for common spatial operations.

Section 26.4 is devoted to multimedia database concepts. **Multimedia databases** provide features that allow users to store and query different types of multimedia information, which includes **images** (such as pictures and drawings), **video clips** (such as movies, newsreels, and home videos), **audio clips** (such as songs, phone messages, and speeches), and **documents** (such as books and articles). We discuss automatic analysis of images, object recognition in images, and semantic tagging of images.

In Section 26.5, we discuss deductive databases,<sup>1</sup> an area that is at the intersection of databases, logic, and artificial intelligence or knowledge bases. A **deductive database system** includes capabilities to define (**deductive**) **rules**, which can deduce or infer additional information from the facts that are stored in a database. Because part of the theoretical foundation for some deductive database systems is mathematical logic, such rules are often referred to as **logic databases**. Other types of systems, referred to as **expert database systems** or **knowledge-based systems**, also incorporate reasoning and inferencing capabilities; such systems use techniques

---

<sup>1</sup>Section 26.5 is a summary of Deductive Databases. The full chapter from the third edition, which provides a more comprehensive introduction, is available on the book's Web site.

that were developed in the field of artificial intelligence, including semantic networks, frames, production systems, or rules for capturing domain-specific knowledge. Section 26.6 summarizes the chapter.

Readers may choose to peruse the particular topics they are interested in, as the sections in this chapter are practically independent of one another.

## 26.1 Active Database Concepts and Triggers

Rules that specify actions that are automatically triggered by certain events have been considered important enhancements to database systems for quite some time. In fact, the concept of **triggers**—a technique for specifying certain types of active rules—has existed in early versions of the SQL specification for relational databases, and triggers are now part of the SQL-99 and later standards. Commercial relational DBMSs—such as Oracle, DB2, and Microsoft SQLServer—have various versions of triggers available. However, much research into what a general model for active databases should look like has been done since the early models of triggers were proposed. In Section 26.1.1, we will present the general concepts that have been proposed for specifying rules for active databases. We will use the syntax of the Oracle commercial relational DBMS to illustrate these concepts with specific examples, since Oracle triggers are close to the way rules are specified in the SQL standard. Section 26.1.2 will discuss some general design and implementation issues for active databases. We give examples of how active databases are implemented in the STARBURST experimental DBMS in Section 26.1.3, since STARBURST provides for many of the concepts of generalized active databases within its framework. Section 26.1.4 discusses possible applications of active databases. Finally, Section 26.1.5 describes how triggers are declared in the SQL-99 standard.

### 26.1.1 Generalized Model for Active Databases and Oracle Triggers

The model that has been used to specify active database rules is referred to as the **event-condition-action (ECA)** model. A rule in the ECA model has three components:

1. The **event(s)** that triggers the rule: These events are usually database update operations that are explicitly applied to the database. However, in the general model, they could also be temporal events<sup>2</sup> or other kinds of external events.
2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event

---

<sup>2</sup>An example would be a temporal event specified as a periodic time, such as: Trigger this rule every day at 5:30 a.m.

**EMPLOYEE**

Name	<u>Ssn</u>	Salary	Dno	Supervisor_ssn
------	------------	--------	-----	----------------

**Figure 26.1**

A simplified COMPANY database used for active rule examples.

**DEPARTMENT**

Dname	<u>Dno</u>	Total_sal	Manager_ssn
-------	------------	-----------	-------------

occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed.

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

Let us consider some examples to illustrate these concepts. The examples are based on a much simplified variation of the COMPANY database application from Figure 5.5 and are shown in Figure 26.1, with each employee having a name (Name), Social Security number (Ssn), salary (Salary), department to which she is currently assigned (Dno, a foreign key to DEPARTMENT), and a direct supervisor (Supervisor\_ssn, a (recursive) foreign key to EMPLOYEE). For this example, we assume that NULL is allowed for Dno, indicating that an employee may be temporarily unassigned to any department. Each department has a name (Dname), number (Dno), the total salary of all employees assigned to the department (Total\_sal), and a manager (Manager\_ssn, which is a foreign key to EMPLOYEE).

Notice that the Total\_sal attribute is really a derived attribute whose value should be the sum of the salaries of all employees who are assigned to the particular department. Maintaining the correct value of such a derived attribute can be done via an active rule. First we have to determine the **events** that *may cause* a change in the value of Total\_sal, which are as follows:

1. Inserting (one or more) new employee tuples
2. Changing the salary of (one or more) existing employees
3. Changing the assignment of existing employees from one department to another
4. Deleting (one or more) employee tuples

In the case of event 1, we only need to recompute Total\_sal if the new employee is immediately assigned to a department—that is, if the value of the Dno attribute for the new employee tuple is not NULL (assuming NULL is allowed for Dno). Hence, this would be the **condition** to be checked. A similar condition could be checked for event 2 (and 4) to determine whether the employee whose salary is changed (or who is being deleted) is currently assigned to a department. For event 3, we will always execute an action to maintain the value of Total\_sal correctly, so no condition is needed (the action is always executed).

The **action** for events 1, 2, and 4 is to automatically update the value of `Total_sal` for the employee's department to reflect the newly inserted, updated, or deleted employee's salary. In the case of event 3, a twofold action is needed: one to update the `Total_sal` of the employee's old department and the other to update the `Total_sal` of the employee's new department.

The four active rules (or triggers) R1, R2, R3, and R4—corresponding to the above situation—can be specified in the notation of the Oracle DBMS as shown in Figure 26.2(a). Let us consider rule R1 to illustrate the syntax of creating triggers in Oracle. The `CREATE TRIGGER` statement specifies a trigger (or active rule) name—`Total_sal1` for R1. The `AFTER` clause specifies that the rule will be triggered *after* the events that trigger the rule occur. The triggering events—an insert of a new employee in this example—are specified following the `AFTER` keyword.<sup>3</sup>

The `ON` clause specifies the relation on which the rule is specified—`EMPLOYEE` for R1. The *optional* keywords `FOR EACH ROW` specify that the rule will be triggered *once for each row* that is affected by the triggering event.<sup>4</sup>

The *optional* `WHEN` clause is used to specify any conditions that need to be checked after the rule is triggered, but before the action is executed. Finally, the action(s) to be taken is (are) specified as a PL/SQL block, which typically contains one or more SQL statements or calls to execute external procedures.

The four triggers (active rules) R1, R2, R3, and R4 illustrate a number of features of active rules. First, the basic **events** that can be specified for triggering the rules are the standard SQL update commands: `INSERT`, `DELETE`, and `UPDATE`. They are specified by the keywords **`INSERT`**, **`DELETE`**, and **`UPDATE`** in Oracle notation. In the case of `UPDATE`, one may specify the attributes to be updated—for example, by writing **`UPDATE OF Salary, Dno`**. Second, the rule designer needs to have a way to refer to the tuples that have been inserted, deleted, or modified by the triggering event. The keywords **`NEW`** and **`OLD`** are used in Oracle notation; `NEW` is used to refer to a newly inserted or newly updated tuple, whereas `OLD` is used to refer to a deleted tuple or to a tuple before it was updated.

Thus, rule R1 is triggered after an `INSERT` operation is applied to the `EMPLOYEE` relation. In R1, the condition (**`NEW.Dno IS NOT NULL`**) is checked, and if it evaluates to true, meaning that the newly inserted employee tuple is related to a department, then the action is executed. The action updates the `DEPARTMENT` tuple(s) related to the newly inserted employee by adding their salary (**`NEW.Salary`**) to the `Total_sal` attribute of their related department.

Rule R2 is similar to R1, but it is triggered by an `UPDATE` operation that updates the `SALARY` of an employee rather than by an `INSERT`. Rule R3 is triggered by an update to the `Dno` attribute of `EMPLOYEE`, which signifies changing an employee's assignment from one department to another. There is no condition to check in R3, so the

---

<sup>3</sup>As we will see, it is also possible to specify `BEFORE` instead of `AFTER`, which indicates that the rule is triggered *before the triggering event is executed*.

<sup>4</sup>Again, we will see that an alternative is to trigger the rule *only once* even if multiple rows (tuples) are affected by the triggering event.



**Figure 26.2**

Specifying active rules as triggers in Oracle notation. (a) Triggers for automatically maintaining the consistency of Total\_sal of DEPARTMENT.

(b) Trigger for comparing an employee's salary with that of his or her supervisor.

```
(a) R1: CREATE TRIGGER Total_sal1
        AFTER INSERT ON EMPLOYEE
        FOR EACH ROW
        WHEN ( NEW.Dno IS NOT NULL )
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal + NEW.Salary
            WHERE Dno = NEW.Dno;

R2: CREATE TRIGGER Total_sal2
        AFTER UPDATE OF Salary ON EMPLOYEE
        FOR EACH ROW
        WHEN ( NEW.Dno IS NOT NULL )
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal + NEW.Salary - OLD.Salary
            WHERE Dno = NEW.Dno;

R3: CREATE TRIGGER Total_sal3
        AFTER UPDATE OF Dno ON EMPLOYEE
        FOR EACH ROW
        BEGIN
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal + NEW.Salary
            WHERE Dno = NEW.Dno;
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal - OLD.Salary
            WHERE Dno = OLD.Dno;
        END;

R4: CREATE TRIGGER Total_sal4
        AFTER DELETE ON EMPLOYEE
        FOR EACH ROW
        WHEN ( OLD.Dno IS NOT NULL )
            UPDATE DEPARTMENT
            SET Total_sal = Total_sal - OLD.Salary
            WHERE Dno = OLD.Dno;

(b) R5: CREATE TRIGGER Inform_supervisor1
        BEFORE INSERT OR UPDATE OF Salary, Supervisor_ssn
        ON EMPLOYEE
        FOR EACH ROW
        WHEN ( NEW.Salary > ( SELECT Salary FROM EMPLOYEE
                               WHERE Ssn = NEW.Supervisor_ssn ) )
            inform_supervisor(NEW.Supervisor_ssn, NEW.Ssn );
```

action is executed whenever the triggering event occurs. The action updates both the old department and new department of the reassigned employees by adding their salary to `Total_sal` of their *new* department and subtracting their salary from `Total_sal` of their *old* department. Note that this should work even if the value of `Dno` is `NULL`, because in this case no department will be selected for the rule action.<sup>5</sup>

It is important to note the effect of the optional `FOR EACH ROW` clause, which signifies that the rule is triggered separately *for each tuple*. This is known as a **row-level trigger**. If this clause was left out, the trigger would be known as a **statement-level trigger** and would be triggered once for each triggering statement. To see the difference, consider the following update operation, which gives a 10% raise to all employees assigned to department 5. This operation would be an event that triggers rule R2:

```
UPDATE EMPLOYEE
SET      Salary = 1.1 * Salary
WHERE    Dno = 5;
```

Because the above statement could update multiple records, a rule using row-level semantics, such as R2 in Figure 26.2, would be triggered *once for each row*, whereas a rule using statement-level semantics is triggered *only once*. The Oracle system allows the user to choose which of the above options is to be used for each rule. Including the optional `FOR EACH ROW` clause creates a row-level trigger, and leaving it out creates a statement-level trigger. Note that the keywords `NEW` and `OLD` can only be used with row-level triggers.

As a second example, suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor. Several events can trigger this rule: inserting a new employee, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external procedure `inform_supervisor`,<sup>6</sup> which will notify the supervisor. The rule could then be written as in R5 (see Figure 26.2(b)).

Figure 26.3 shows the syntax for specifying some of the main options available in Oracle triggers. We will describe the syntax for triggers in the SQL-99 standard in Section 26.1.5.

### 26.1.2 Design and Implementation Issues for Active Databases

The previous section gave an overview of some of the main concepts for specifying active rules. In this section, we discuss some additional issues concerning how rules are designed and implemented. The first issue concerns activation,

---

<sup>5</sup>R1, R2, and R4 can also be written without a condition. However, it may be more efficient to execute them with the condition since the action is not invoked unless it is required.

<sup>6</sup>Assuming that an appropriate external procedure has been declared. This is a feature that is available in SQL-99 and later standards.

```

<trigger>          ::= CREATE TRIGGER <trigger name>
                    ( AFTER | BEFORE ) <triggering events> ON <table name>
                    [ FOR EACH ROW ]
                    [ WHEN <condition> ]
                    <trigger actions> ;
<triggering events> ::= <trigger event> {OR <trigger event> }
<trigger event>     ::= INSERT | DELETE | UPDATE [ OF <column name> { , <column name> } ]
<trigger action>    ::= <PL/SQL block>

```

**Figure 26.3**

A syntax summary for specifying triggers in the Oracle system (main options only).

---

deactivation, and grouping of rules. In addition to creating rules, an active database system should allow users to *activate*, *deactivate*, and *drop* rules by referring to their rule names. A **deactivated rule** will not be triggered by the triggering event. This feature allows users to selectively deactivate rules for certain periods of time when they are not needed. The **activate command** will make the rule active again. The **drop command** deletes the rule from the system. Another option is to group rules into named **rule sets**, so the whole set of rules can be activated, deactivated, or dropped. It is also useful to have a command that can trigger a rule or rule set via an explicit **PROCESS RULES** command issued by the user.

The second issue concerns whether the triggered action should be executed *before*, *after*, *instead of*, or *concurrently with* the triggering event. A **before trigger** executes the trigger before executing the event that caused the trigger. It can be used in applications such as checking for constraint violations. An **after trigger** executes the trigger after executing the event, and it can be used in applications such as maintaining derived data and monitoring for specific events and conditions. An **instead of trigger** executes the trigger instead of executing the event, and it can be used in applications such as executing corresponding updates on base relations in response to an event that is an update of a view.

A related issue is whether the action being executed should be considered as a *separate transaction* or whether it should be part of the same transaction that triggered the rule. We will try to categorize the various options. It is important to note that not all options may be available for a particular active database system. In fact, most commercial systems are *limited to one or two of the options* that we will now discuss.

Let us assume that the triggering event occurs as part of a transaction execution. We should first consider the various options for how the triggering event is related to the evaluation of the rule's condition. The rule *condition evaluation* is also known as **rule consideration**, since the action is to be executed only after considering whether the condition evaluates to true or false. There are three main possibilities for rule consideration:

1. **Immediate consideration.** The condition is evaluated as part of the same transaction as the triggering event and is evaluated *immediately*. This case can be further categorized into three options:
  - Evaluate the condition *before* executing the triggering event.
  - Evaluate the condition *after* executing the triggering event.
  - Evaluate the condition *instead of* executing the triggering event.
2. **Deferred consideration.** The condition is evaluated at the end of the transaction that included the triggering event. In this case, there could be many triggered rules waiting to have their conditions evaluated.
3. **Detached consideration.** The condition is evaluated as a separate transaction, spawned from the triggering transaction.

The next set of options concerns the relationship between evaluating the rule condition and *executing* the rule action. Here, again, three options are possible: **immediate**, **deferred**, or **detached** execution. Most active systems use the first option. That is, as soon as the condition is evaluated, if it returns true, the action is *immediately* executed.

The Oracle system (see Section 26.1.1) uses the *immediate consideration* model, but it allows the user to specify for each rule whether the *before* or *after* option is to be used with immediate condition evaluation. It also uses the *immediate execution* model. The STARBURST system (see Section 26.1.3) uses the *deferred consideration* option, meaning that all rules triggered by a transaction wait until the triggering transaction reaches its end and issues its COMMIT WORK command before the rule conditions are evaluated.<sup>7</sup>

Another issue concerning active database rules is the distinction between *row-level rules* and *statement-level rules*. Because SQL update statements (which act as triggering events) can specify a set of tuples, one must distinguish between whether the rule should be considered once for the *whole statement* or whether it should be considered separately *for each row* (that is, tuple) affected by the statement. The SQL-99 standard (see Section 26.1.5) and the Oracle system (see Section 26.1.1) allow the user to choose which of the options is to be used for each rule, whereas STARBURST uses statement-level semantics only. We will give examples of how statement-level triggers can be specified in Section 26.1.3.

One of the difficulties that may have limited the widespread use of active rules, in spite of their potential to simplify database and software development, is that there are no easy-to-use techniques for designing, writing, and verifying rules. For example, it is difficult to verify that a set of rules is **consistent**, meaning that two or more rules in the set do not contradict one another. It is also difficult to guarantee **termination** of a set of rules under all circumstances. To illustrate the termination problem briefly, consider the rules in Figure 26.4. Here, rule R1 is triggered by an INSERT event on TABLE1 and its action includes an update event on Attribute1 of

---

<sup>7</sup>STARBURST also allows the user to start rule consideration explicitly via a PROCESS RULES command.

```

R1: CREATE TRIGGER T1
    AFTER INSERT ON TABLE1
    FOR EACH ROW
    UPDATE TABLE2
    SET Attribute1 = ... ;

R2: CREATE TRIGGER T2
    AFTER UPDATE OF Attribute1 ON TABLE2
    FOR EACH ROW
    INSERT INTO TABLE1 VALUES ( ... );

```

**Figure 26.4**

An example to illustrate the termination problem for active rules.

TABLE2. However, rule R2's triggering event is an UPDATE event on Attribute1 of TABLE2, and its action includes an INSERT event on TABLE1. In this example, it is easy to see that these two rules can trigger one another indefinitely, leading to non-termination. However, if dozens of rules are written, it is very difficult to determine whether termination is guaranteed or not.

If active rules are to reach their potential, it is necessary to develop tools for the design, debugging, and monitoring of active rules that can help users design and debug their rules.

### 26.1.3 Examples of Statement-Level Active Rules in STARBURST

We now give some examples to illustrate how rules can be specified in the STARBURST experimental DBMS. This will allow us to demonstrate how statement-level rules can be written, since these are the only types of rules allowed in STARBURST.

The three active rules R1S, R2S, and R3S in Figure 26.5 correspond to the first three rules in Figure 26.2, but they use STARBURST notation and statement-level semantics. We can explain the rule structure using rule R1S. The CREATE RULE statement specifies a rule name—Total\_sal1 for R1S. The ON clause specifies the relation on which the rule is specified—EMPLOYEE for R1S. The WHEN clause is used to specify the **events** that trigger the rule.<sup>8</sup> The *optional* IF clause is used to specify any **conditions** that need to be checked. Finally, the THEN clause is used to specify the **actions** to be taken, which are typically one or more SQL statements.

In STARBURST, the basic events that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. These are specified by the keywords **INSERTED**, **DELETED**, and **UPDATED** in STARBURST notation. Second, the rule designer needs to have a way to refer to the tuples that have been modified. The keywords **INSERTED**, **DELETED**, **NEW-UPDATED**, and **OLD-UPDATED** are used in STARBURST notation to refer to four **transition tables** (relations) that include the newly inserted tuples, the deleted tuples, the updated

<sup>8</sup>Note that the WHEN keyword specifies *events* in STARBURST but is used to specify the rule *condition* in SQL and Oracle triggers.

```

R1S: CREATE RULE Total_sal1 ON EMPLOYEE
      WHEN INSERTED
      IF EXISTS ( SELECT * FROM INSERTED WHERE Dno IS NOT NULL)
      THEN UPDATE DEPARTMENT AS D
                SET D.Total_sal = D.Total_sal +
                    ( SELECT SUM (I.Salary) FROM INSERTED AS I WHERE D.Dno = I.Dno )
                WHERE D.Dno IN ( SELECT Dno FROM INSERTED );

R2S: CREATE RULE Total_sal2 ON EMPLOYEE
      WHEN UPDATED ( Salary )
      IF EXISTS ( SELECT * FROM NEW-UPDATED WHERE Dno IS NOT NULL)
      OR EXISTS ( SELECT * FROM OLD-UPDATED WHERE Dno IS NOT NULL)
      THEN UPDATE DEPARTMENT AS D
                SET D.Total_sal = D.Total_sal +
                    ( SELECT SUM (N.Salary) FROM NEW-UPDATED AS N
                      WHERE D.Dno = N.Dno ) -
                    ( SELECT SUM (O.Salary) FROM OLD-UPDATED AS O
                      WHERE D.Dno = O.Dno )
                WHERE D.Dno IN ( SELECT Dno FROM NEW-UPDATED ) OR
                      D.Dno IN ( SELECT Dno FROM OLD-UPDATED );

R3S: CREATE RULE Total_sal3 ON EMPLOYEE
      WHEN UPDATED ( Dno )
      THEN UPDATE DEPARTMENT AS D
                SET D.Total_sal = D.Total_sal +
                    ( SELECT SUM (N.Salary) FROM NEW-UPDATED AS N
                      WHERE D.Dno = N.Dno )
                WHERE D.Dno IN ( SELECT Dno FROM NEW-UPDATED );
      UPDATE DEPARTMENT AS D
      SET D.Total_sal = Total_sal -
          ( SELECT SUM (O.Salary) FROM OLD-UPDATED AS O
            WHERE D.Dno = O.Dno )
      WHERE D.Dno IN ( SELECT Dno FROM OLD-UPDATED );

```

**Figure 26.5**

Active rules using statement-level semantics in STARBURST notation.

tuples *before* they were updated, and the updated tuples *after* they were updated, respectively. Obviously, depending on the triggering events, only some of these transition tables may be available. The rule writer can refer to these tables when writing the condition and action parts of the rule. Transition tables contain tuples of the same type as those in the relation specified in the ON clause of the rule—for R1S, R2S, and R3S, this is the EMPLOYEE relation.

In statement-level semantics, the rule designer can only refer to the transition tables as a whole and the rule is triggered only once, so the rules must be written differently than for row-level semantics. Because multiple employee tuples may be

inserted in a single insert statement, we have to check if *at least one* of the newly inserted employee tuples is related to a department. In R1S, the condition

**EXISTS (SELECT \* FROM INSERTED WHERE Dno IS NOT NULL )**

is checked, and if it evaluates to true, then the action is executed. The action updates in a single statement the DEPARTMENT tuple(s) related to the newly inserted employee(s) by adding their salaries to the Total\_sal attribute of each related department. Because more than one newly inserted employee may belong to the same department, we use the SUM aggregate function to ensure that all their salaries are added.

Rule R2S is similar to R1S, but is triggered by an UPDATE operation that updates the salary of one or more employees rather than by an INSERT. Rule R3S is triggered by an update to the Dno attribute of EMPLOYEE, which signifies changing one or more employees' assignment from one department to another. There is no condition in R3S, so the action is executed whenever the triggering event occurs.<sup>9</sup> The action updates both the old department(s) and new department(s) of the reassigned employees by adding their salary to Total\_sal of each *new* department and subtracting their salary from Total\_sal of each *old* department.

In our example, it is more complex to write the statement-level rules than the row-level rules, as can be illustrated by comparing Figures 26.2 and 26.5. However, this is not a general rule, and other types of active rules may be easier to specify when using statement-level notation than when using row-level notation.

The execution model for active rules in STARBURST uses **deferred consideration**. That is, all the rules that are triggered within a transaction are placed in a set—called the **conflict set**—which is not considered for evaluation of conditions and execution until the transaction ends (by issuing its COMMIT WORK command). STARBURST also allows the user to explicitly start rule consideration in the middle of a transaction via an explicit PROCESS RULES command. Because multiple rules must be evaluated, it is necessary to specify an order among the rules. The syntax for rule declaration in STARBURST allows the specification of *ordering* among the rules to instruct the system about the order in which a set of rules should be considered.<sup>10</sup> Additionally, the transition tables—INSERTED, DELETED, NEW-UPDATED, and OLD-UPDATED—contain the *net effect* of all the operations within the transaction that affected each table, since multiple operations may have been applied to each table during the transaction.

### 26.1.4 Potential Applications for Active Databases

We now briefly discuss some of the potential applications of active rules. Obviously, one important application is to allow **notification** of certain conditions that

<sup>9</sup>As in the Oracle examples, rules R1S and R2S can be written without a condition. However, it may be more efficient to execute them with the condition since the action is not invoked unless it is required.

<sup>10</sup>If no order is specified between a pair of rules, the system default order is based on placing the rule declared first ahead of the other rule.

occur. For example, an active database may be used to monitor, say, the temperature of an industrial furnace. The application can periodically insert in the database the temperature reading records directly from temperature sensors, and active rules can be written that are triggered whenever a temperature record is inserted, with a condition that checks if the temperature exceeds the danger level and results in the action to raise an alarm.

Active rules can also be used to **enforce integrity constraints** by specifying the types of events that may cause the constraints to be violated and then evaluating appropriate conditions that check whether the constraints are actually violated by the event or not. Hence, complex application constraints, often known as **business rules**, may be enforced that way. For example, in the UNIVERSITY database application, one rule may monitor the GPA of students whenever a new grade is entered, and it may alert the advisor if the GPA of a student falls below a certain threshold; another rule may check that course prerequisites are satisfied before allowing a student to enroll in a course; and so on.

Other applications include the automatic **maintenance of derived data**, such as the examples of rules R1 through R4 that maintain the derived attribute `Total_sal` whenever individual employee tuples are changed. A similar application is to use active rules to maintain the consistency of **materialized views** (see Section 5.3) whenever the base relations are modified. Alternately, an update operation specified on a view can be a triggering event, which can be converted to updates on the base relations by using an *instead of* trigger. These applications are also relevant to the new data warehousing technologies (see Chapter 29). A related application maintains that **replicated tables** are consistent by specifying rules that modify the replicas whenever the master table is modified.

### 26.1.5 Triggers in SQL-99

Triggers in the SQL-99 and later standards are similar to the examples we discussed in Section 26.1.1, with some minor syntactic differences. The basic **events** that can be specified for triggering the rules are the standard SQL update commands: INSERT, DELETE, and UPDATE. In the case of UPDATE, one may specify the attributes to be updated. Both row-level and statement-level triggers are allowed, indicated in the trigger by the clauses FOR EACH ROW and FOR EACH STATEMENT, respectively. One syntactic difference is that the trigger may specify particular tuple variable names for the old and new tuples instead of using the keywords NEW and OLD, as shown in Figure 26.1. Trigger T1 in Figure 26.6 shows how the row-level trigger R2 from Figure 26.1(a) may be specified in SQL-99. Inside the REFERENCING clause, we named tuple variables (aliases) O and N to refer to the OLD tuple (before modification) and NEW tuple (after modification), respectively. Trigger T2 in Figure 26.6 shows how the statement-level trigger R2S from Figure 26.5 may be specified in SQL-99. For a statement-level trigger, the REFERENCING clause is used to refer to the table of all new tuples (newly inserted or newly updated) as N, whereas the table of all old tuples (deleted tuples or tuples before they were updated) is referred to as O.



```

T1: CREATE TRIGGER Total_sal1
AFTER UPDATE OF Salary ON EMPLOYEE
REFERENCING OLD ROW AS O, NEW ROW AS N
FOR EACH ROW
WHEN ( N.Dno IS NOT NULL )
UPDATE DEPARTMENT
SET Total_sal = Total_sal + N.salary - O.salary
WHERE Dno = N.Dno;

T2: CREATE TRIGGER Total_sal2
AFTER UPDATE OF Salary ON EMPLOYEE
REFERENCING OLD TABLE AS O, NEW TABLE AS N
FOR EACH STATEMENT
WHEN EXISTS ( SELECT * FROM N WHERE N.Dno IS NOT NULL ) OR
EXISTS ( SELECT * FROM O WHERE O.Dno IS NOT NULL )
UPDATE DEPARTMENT AS D
SET D.Total_sal = D.Total_sal
+ ( SELECT SUM (N.Salary) FROM N WHERE D.Dno=N.Dno )
- ( SELECT SUM (O.Salary) FROM O WHERE D.Dno=O.Dno )
WHERE Dno IN ( ( SELECT Dno FROM N ) UNION ( SELECT Dno FROM O ) );

```

**Figure 26.6**

Trigger T1 illustrating the syntax for defining triggers in SQL-99.

## 26.2 Temporal Database Concepts

Temporal databases, in the broadest sense, encompass all database applications that require some aspect of time when organizing their information. Hence, they provide a good example to illustrate the need for developing a set of unifying concepts for application developers to use. Temporal database applications have been developed since the early days of database usage. However, in creating these applications, it is mainly left to the application designers and developers to discover, design, program, and implement the temporal concepts they need. There are many examples of applications where some aspect of time is needed to maintain the information in a database. These include *healthcare*, where patient histories need to be maintained; *insurance*, where claims and accident histories are required as well as information about the times when insurance policies are in effect; *reservation systems* in general (hotel, airline, car rental, train, and so on), where information on the dates and times when reservations are in effect are required; *scientific databases*, where data collected from experiments includes the time when each data is measured; and so on. Even the two examples used in this book may be easily expanded into temporal applications. In the COMPANY database, we may wish to keep SALARY, JOB, and PROJECT histories on each employee. In the UNIVERSITY database, time is already included in the SEMESTER and YEAR of each SECTION of a COURSE, the grade history of a STUDENT, and the information on research grants. In fact, it is realistic to conclude that the majority of database applications have some temporal information. However, users often attempt to simplify or ignore temporal aspects because of the complexity that they add to their applications.

In this section, we will introduce some of the concepts that have been developed to deal with the complexity of temporal database applications. Section 26.2.1 gives an overview of how time is represented in databases, the different types of temporal information, and some of the different dimensions of time that may be needed. Section 26.2.2 discusses how time can be incorporated into relational databases. Section 26.2.3 gives some additional options for representing time that are possible in database models that allow complex-structured objects, such as object databases. Section 26.2.4 introduces operations for querying temporal databases and gives a brief overview of the TSQL2 language, which extends SQL with temporal concepts. Section 26.2.5 focuses on time series data, which is a type of temporal data that is very important in practice.

### 26.2.1 Time Representation, Calendars, and Time Dimensions

For temporal databases, time is considered to be an *ordered sequence* of **points** in some **granularity** that is determined by the application. For example, suppose that some temporal application never requires time units that are less than one second. Then, each time point represents one second using this granularity. In reality, each second is a (short) *time duration*, not a point, since it may be further divided into milliseconds, microseconds, and so on. Temporal database researchers have used the term **chronon** instead of *point* to describe this minimal granularity for a particular application. The main consequence of choosing a minimum granularity—say, one second—is that events occurring within the same second will be considered to be *simultaneous events*, even though in reality they may not be.

Because there is no known beginning or ending of time, one needs a reference point from which to measure specific time points. Various calendars are used by various cultures (such as Gregorian (Western), Chinese, Islamic, Hindu, Jewish, Coptic, and so on) with different reference points. A **calendar** organizes time into different time units for convenience. Most calendars group 60 seconds into a minute, 60 minutes into an hour, 24 hours into a day (based on the physical time of earth's rotation around its axis), and 7 days into a week. Further groupings of days into months and months into years either follow solar or lunar natural phenomena and are generally irregular. In the Gregorian calendar, which is used in most Western countries, days are grouped into months that are 28, 29, 30, or 31 days, and 12 months are grouped into a year. Complex formulas are used to map the different time units to one another.

In SQL2, the temporal data types (see Chapter 4) include DATE (specifying Year, Month, and Day as YYYY-MM-DD), TIME (specifying Hour, Minute, and Second as HH:MM:SS), TIMESTAMP (specifying a Date/Time combination, with options for including subsecond divisions if they are needed), INTERVAL (a relative time duration, such as 10 days or 250 minutes), and PERIOD (an *anchored* time duration with a fixed starting point, such as the 10-day period from January 1, 2009, to January 10, 2009, inclusive).<sup>11</sup>

---

<sup>11</sup>Unfortunately, the terminology has not been used consistently. For example, the term *interval* is often used to denote an anchored duration. For consistency, we will use the SQL terminology.

**Event Information versus Duration (or State) Information.** A temporal database will store information concerning when certain events occur, or when certain facts are considered to be true. There are several different types of temporal information. **Point events** or **facts** are typically associated in the database with a **single time point** in some granularity. For example, a bank deposit event may be associated with the timestamp when the deposit was made, or the total monthly sales of a product (fact) may be associated with a particular month (say, February 2010). Note that even though such events or facts may have different granularities, each is still associated with a *single time value* in the database. This type of information is often represented as **time series data**, as we will discuss in Section 26.2.5. **Duration events** or **facts**, on the other hand, are associated with a specific **time period** in the database.<sup>12</sup> For example, an employee may have worked in a company from August 15, 2003 until November 20, 2008.

A **time period** is represented by its **start** and **end time points** [START-TIME, ENDTIME]. For example, the above period is represented as [2003-08-15, 2008-11-20]. Such a time period is often interpreted to mean the *set of all time points* from start-time to end-time, inclusive, in the specified granularity. Hence, assuming day granularity, the period [2003-08-15, 2008-11-20] represents the set of all days from August 15, 2003, until November 20, 2008, inclusive.<sup>13</sup>

**Valid Time and Transaction Time Dimensions.** Given a particular event or fact that is associated with a particular time point or time period in the database, the association may be interpreted to mean different things. The most natural interpretation is that the associated time is the time that the event occurred, or the period during which the fact was considered to be true *in the real world*. If this interpretation is used, the associated time is often referred to as the **valid time**. A temporal database using this interpretation is called a **valid time database**.

However, a different interpretation can be used, where the associated time refers to the time when the information was actually stored in the database; that is, it is the value of the system time clock when the information is valid *in the system*.<sup>14</sup> In this case, the associated time is called the **transaction time**. A temporal database using this interpretation is called a **transaction time database**.

Other interpretations can also be intended, but these are considered to be the most common ones, and they are referred to as **time dimensions**. In some applications, only one of the dimensions is needed and in other cases both time dimensions are required, in which case the temporal database is called a **bitemporal database**. If

<sup>12</sup>This is the same as an *anchored duration*. It has also been frequently called a *time interval*, but to avoid confusion we will use *period* to be consistent with SQL terminology.

<sup>13</sup>The representation [2003-08-15, 2008-11-20] is called a *closed interval* representation. One can also use an *open interval*, denoted [2003-08-15, 2008-11-21), where the set of points does not include the end point. Although the latter representation is sometimes more convenient, we shall use closed intervals except where indicated.

<sup>14</sup>The explanation is more involved, as we will see in Section 26.2.3.

other interpretations are intended for time, the user can define the semantics and program the applications appropriately, and this interpretation of time is called a **user-defined time**.

The next section shows how these concepts can be incorporated into relational databases, and Section 26.2.3 shows an approach to incorporate temporal concepts into object databases.

### 26.2.2 Incorporating Time in Relational Databases Using Tuple Versioning

**Valid Time Relations.** Let us now see how the different types of temporal databases may be represented in the relational model. First, suppose that we would like to include the history of changes as they occur in the real world. Consider again the database in Figure 26.1, and let us assume that, for this application, the granularity is day. Then, we could convert the two relations EMPLOYEE and DEPARTMENT into **valid time relations** by adding the attributes Vst (Valid Start Time) and Vet (Valid End Time), whose data type is DATE in order to provide day granularity. This is shown in Figure 26.7(a), where the relations have been renamed EMP\_VT and DEPT\_VT, respectively.

Consider how the EMP\_VT relation differs from the nontemporal EMPLOYEE relation (Figure 26.1).<sup>15</sup> In EMP\_VT, each tuple V represents a **version** of an employee's

**(a) EMP\_VT**

Name	<u>Ssn</u>	Salary	Dno	Supervisor_ssn	<u>Vst</u>	Vet
------	------------	--------	-----	----------------	------------	-----

**DEPT\_VT**

Dname	<u>Dno</u>	Total_sal	Manager_ssn	<u>Vst</u>	Vet
-------	------------	-----------	-------------	------------	-----

**(b) EMP\_TT**

Name	<u>Ssn</u>	Salary	Dno	Supervisor_ssn	<u>Tst</u>	Tet
------	------------	--------	-----	----------------	------------	-----

**DEPT\_TT**

Dname	<u>Dno</u>	Total_sal	Manager_ssn	<u>Tst</u>	Tet
-------	------------	-----------	-------------	------------	-----

**(c) EMP\_BT**

Name	<u>Ssn</u>	Salary	Dno	Supervisor_ssn	<u>Vst</u>	Vet	<u>Tst</u>	Tet
------	------------	--------	-----	----------------	------------	-----	------------	-----

**DEPT\_BT**

Dname	<u>Dno</u>	Total_sal	Manager_ssn	<u>Vst</u>	Vet	<u>Tst</u>	Tet
-------	------------	-----------	-------------	------------	-----	------------	-----

**Figure 26.7**

Different types of temporal relational databases. (a) Valid time database schema. (b) Transaction time database schema. (c) Bitemporal database schema.

<sup>15</sup>A nontemporal relation is also called a **snapshot relation** because it shows only the *current snapshot* or *current state* of the database.

information that is valid (in the real world) only during the time period [V.Vst, V.Vet], whereas in EMPLOYEE each tuple represents only the current state or current version of each employee. In EMP\_VT, the **current version** of each employee typically has a special value, *now*, as its valid end time. This special value, *now*, is a **temporal variable** that implicitly represents the current time as time progresses. The nontemporal EMPLOYEE relation would only include those tuples from the EMP\_VT relation whose Vet is *now*.

Figure 26.8 shows a few tuple versions in the valid-time relations EMP\_VT and DEPT\_VT. There are two versions of Smith, three versions of Wong, one version of Brown, and one version of Narayan. We can now see how a valid time relation should behave when information is changed. Whenever one or more attributes of an employee are **updated**, rather than actually overwriting the old values, as would happen in a nontemporal relation, the system should create a new version and **close** the current version by changing its Vet to the end time. Hence, when the user issued the command to update the salary of Smith effective on June 1, 2003, to \$30000, the second version of Smith was created (see Figure 26.8). At the time of this update, the first version of Smith was the current version, with *now* as its Vet, but after the update *now* was changed to May 31, 2003 (one less than June 1, 2003, in day granularity), to indicate that the version has become a **closed** or **history version** and that the new (second) version of Smith is now the current one.

It is important to note that in a valid time relation, the user must generally provide the valid time of an update. For example, the salary update of Smith may have been

**Figure 26.8**

Some tuple versions in the valid time relations EMP\_VT and DEPT\_VT.

#### EMP\_VT

Name	Ssn	Salary	Dno	Supervisor_ssn	Vst	Vet
Smith	123456789	25000	5	333445555	2002-06-15	2003-05-31
Smith	123456789	30000	5	333445555	2003-06-01	Now
Wong	333445555	25000	4	999887777	1999-08-20	2001-01-31
Wong	333445555	30000	5	999887777	2001-02-01	2002-03-31
Wong	333445555	40000	5	888665555	2002-04-01	Now
Brown	222447777	28000	4	999887777	2001-05-01	2002-08-10
Narayan	666884444	38000	5	333445555	2003-08-01	Now

...

#### DEPT\_VT

Dname	Dno	Manager_ssn	Vst	Vet
Research	5	888665555	2001-09-20	2002-03-31
Research	5	333445555	2002-04-01	Now

...

entered in the database on May 15, 2003, at 8:52:12 a.m., say, even though the salary change in the real world is effective on June 1, 2003. This is called a **proactive update**, since it is applied to the database *before* it becomes effective in the real world. If the update is applied to the database *after* it becomes effective in the real world, it is called a **retroactive update**. An update that is applied at the same time as it becomes effective is called a **simultaneous update**.

The action that corresponds to **deleting** an employee in a nontemporal database would typically be applied to a valid time database by *closing the current version* of the employee being deleted. For example, if Smith leaves the company effective January 19, 2004, then this would be applied by changing Vet of the current version of Smith from *now* to 2004-01-19. In Figure 26.8, there is no current version for Brown, because he presumably left the company on 2002-08-10 and was *logically deleted*. However, because the database is temporal, the old information on Brown is still there.

The operation to **insert** a new employee would correspond to *creating the first tuple version* for that employee and making it the current version, with the Vst being the effective (real world) time when the employee starts work. In Figure 26.7, the tuple on Narayan illustrates this, since the first version has not been updated yet.

Notice that in a valid time relation, the *nontemporal key*, such as Ssn in EMPLOYEE, is no longer unique in each tuple (version). The new relation key for EMP\_VT is a combination of the nontemporal key and the valid start time attribute Vst,<sup>16</sup> so we use (Ssn, Vst) as primary key. This is because, at any point in time, there should be *at most one valid version* of each entity. Hence, the constraint that any two tuple versions representing the same entity should have *nonintersecting valid time periods* should hold on valid time relations. Notice that if the nontemporal primary key value may change over time, it is important to have a unique **surrogate key attribute**, whose value never changes for each real-world entity, in order to relate all versions of the same real-world entity.

Valid time relations basically keep track of the history of changes as they become effective in the *real world*. Hence, if all real-world changes are applied, the database keeps a history of the *real-world states* that are represented. However, because updates, insertions, and deletions may be applied retroactively or proactively, there is no record of the actual *database state* at any point in time. If the actual database states are important to an application, then one should use *transaction time relations*.

**Transaction Time Relations.** In a transaction time database, whenever a change is applied to the database, the actual **timestamp** of the transaction that applied the change (insert, delete, or update) is recorded. Such a database is most useful when changes are applied *simultaneously* in the majority of cases—for example, real-time stock trading or banking transactions. If we convert the nontemporal database in Figure 26.1 into a transaction time database, then the two relations EMPLOYEE and DEPARTMENT are converted into **transaction time relations** by adding the attributes Tst (Transaction Start Time) and Tet (Transaction End Time), whose data

<sup>16</sup>A combination of the nontemporal key and the valid end time attribute **Vet** could also be used.

type is typically **TIMESTAMP**. This is shown in Figure 26.7(b), where the relations have been renamed EMP\_TT and DEPT\_TT, respectively.

In EMP\_TT, each tuple  $V$  represents a *version* of an employee's information that was created at actual time  $V.Tst$  and was (logically) removed at actual time  $V.Tet$  (because the information was no longer correct). In EMP\_TT, the *current version* of each employee typically has a special value, **uc (Until Changed)**, as its transaction end time, which indicates that the tuple represents correct information *until it is changed* by some other transaction.<sup>17</sup> A transaction time database has also been called a **rollback database**,<sup>18</sup> because a user can logically roll back to the actual database state at any past point in time  $T$  by retrieving all tuple versions  $V$  whose transaction time period  $[V.Tst, V.Tet]$  includes time point  $T$ .

**Bitemporal Relations.** Some applications require both valid time and transaction time, leading to **bitemporal relations**. In our example, Figure 26.7(c) shows how the EMPLOYEE and DEPARTMENT nontemporal relations in Figure 26.1 would appear as bitemporal relations EMP\_BT and DEPT\_BT, respectively. Figure 26.9 shows a few tuples in these relations. In these tables, tuples whose transaction end time  $Tet$  is **uc** are the ones representing currently valid information, whereas tuples whose  $Tet$  is an absolute timestamp are tuples that were valid until (just before) that timestamp. Hence, the tuples with **uc** in Figure 26.9 correspond to the valid time tuples in Figure 26.7. The transaction start time attribute  $Tst$  in each tuple is the timestamp of the transaction that created that tuple.

Now consider how an **update operation** would be implemented on a bitemporal relation. In this model of bitemporal databases,<sup>19</sup> *no attributes are physically changed* in any tuple except for the transaction end time attribute  $Tet$  with a value of **uc**.<sup>20</sup> To illustrate how tuples are created, consider the EMP\_BT relation. The *current version*  $V$  of an employee has **uc** in its  $Tet$  attribute and *now* in its  $Vet$  attribute. If some attribute—say, Salary—is updated, then the transaction  $T$  that performs the update should have two parameters: the new value of Salary and the valid time  $VT$  when the new salary becomes effective (in the real world). Assume that  $VT-$  is the time point before  $VT$  in the given valid time granularity and that transaction  $T$  has a timestamp  $TS(T)$ . Then, the following physical changes would be applied to the EMP\_BT table:

1. Make a copy  $V_2$  of the current version  $V$ ; set  $V_2.Vet$  to  $VT-$ ,  $V_2.Tst$  to  $TS(T)$ ,  $V_2.Tet$  to **uc**, and insert  $V_2$  in EMP\_BT;  $V_2$  is a copy of the previous current version  $V$  *after it is closed* at valid time  $VT-$ .

<sup>17</sup>The **uc** variable in transaction time relations corresponds to the *now* variable in valid time relations. However, the semantics are slightly different.

<sup>18</sup>Here, the term *rollback* does not have the same meaning as *transaction rollback* (see Chapter 23) during recovery, where the transaction updates are *physically undone*. Rather, here the updates can be *logically undone*, allowing the user to examine the database as it appeared at a previous time point.

<sup>19</sup>There have been many proposed temporal database models. We describe specific models here as examples to illustrate the concepts.

<sup>20</sup>Some bitemporal models allow the  $Vet$  attribute to be changed also, but the interpretations of the tuples are different in those models.



EMP\_BT

Name	Ssn	Salary	Dno	Supervisor_ssn	Vst	Vet	Tst	Tet
Smith	123456789	25000	5	333445555	2002-06-15	Now	2002-06-08, 13:05:58	2003-06-04,08:56:12
Smith	123456789	25000	5	333445555	2002-06-15	2003-05-31	2003-06-04, 08:56:12	uc
Smith	123456789	30000	5	333445555	2003-06-01	Now	2003-06-04, 08:56:12	uc
Wong	333445555	25000	4	999887777	1999-08-20	Now	1999-08-20, 11:18:23	2001-01-07,14:33:02
Wong	333445555	25000	4	999887777	1999-08-20	2001-01-31	2001-01-07, 14:33:02	uc
Wong	333445555	30000	5	999887777	2001-02-01	Now	2001-01-07, 14:33:02	2002-03-28,09:23:57
Wong	333445555	30000	5	999887777	2001-02-01	2002-03-31	2002-03-28, 09:23:57	uc
Wong	333445555	40000	5	888667777	2002-04-01	Now	2002-03-28, 09:23:57	uc
Brown	222447777	28000	4	999887777	2001-05-01	Now	2001-04-27, 16:22:05	2002-08-12,10:11:07
Brown	222447777	28000	4	999887777	2001-05-01	2002-08-10	2002-08-12, 10:11:07	uc
Narayan	666884444	38000	5	333445555	2003-08-01	Now	2003-07-28, 09:25:37	uc

...

DEPT\_VT

Dname	Dno	Manager_ssn	Vst	Vet	Tst	Tet
Research	5	888665555	2001-09-20	Now	2001-09-15,14:52:12	2001-03-28,09:23:57
Research	5	888665555	2001-09-20	1997-03-31	2002-03-28,09:23:57	uc
Research	5	333445555	2002-04-01	Now	2002-03-28,09:23:57	uc

**Figure 26.9**

Some tuple versions in the bitemporal relations EMP\_BT and DEPT\_BT.

2. Make a copy  $V_3$  of the current version  $V$ ; set  $V_3.Vst$  to  $VT$ ,  $V_3.Vet$  to *now*,  $V_3.Salary$  to the new salary value,  $V_3.Tst$  to  $TS(T)$ ,  $V_3.Tet$  to *uc*, and insert  $V_3$  in EMP\_BT;  $V_3$  represents the new current version.
3. Set  $V.Tet$  to  $TS(T)$  since the current version is no longer representing correct information.

As an illustration, consider the first three tuples  $V_1$ ,  $V_2$ , and  $V_3$  in EMP\_BT in Figure 26.9. Before the update of Smith's salary from 25000 to 30000, only  $V_1$  was in EMP\_BT and it was the current version and its Tet was *uc*. Then, a transaction  $T$  whose timestamp  $TS(T)$  is '2003-06-04,08:56:12' updates the salary to 30000 with the effective valid time of '2003-06-01'. The tuple  $V_2$  is created, which is a copy of  $V_1$  except that its Vet is set to '2003-05-31', one day less than the new valid time, and its Tst is the timestamp of the updating transaction. The tuple  $V_3$  is also created, which has the new salary, its Vst is set to '2003-06-01', and its Tst is also the timestamp of the updating transaction. Finally, the Tet of  $V_1$  is set to the timestamp of the updating transaction, '2003-06-04,08:56:12'. Note that this is a *retroactive update*, since the updating transaction ran on June 4, 2003, but the salary change is effective on June 1, 2003.

Similarly, when Wong's salary and department are updated (at the same time) to 30000 and 5, the updating transaction's timestamp is '2001-01-07,14:33:02' and the effective valid time for the update is '2001-02-01'. Hence, this is a *proactive update* because the transaction ran on January 7, 2001, but the effective date was February 1, 2001. In this case, tuple  $V_4$  is logically replaced by  $V_5$  and  $V_6$ .



Next, let us illustrate how a **delete operation** would be implemented on a bitemporal relation by considering the tuples  $V_9$  and  $V_{10}$  in the EMP\_BT relation of Figure 26.9. Here, employee Brown left the company effective August 10, 2002, and the logical delete is carried out by a transaction  $T$  with  $TS(T) = 2002-08-12, 10:11:07$ . Before this,  $V_9$  was the current version of Brown, and its Tet was  $uc$ . The logical delete is implemented by setting  $V_9.Tet$  to 2002-08-12, 10:11:07 to invalidate it, and creating the *final version*  $V_{10}$  for Brown, with its  $Vet = 2002-08-10$  (see Figure 26.9). Finally, an **insert operation** is implemented by creating the *first version* as illustrated by  $V_{11}$  in the EMP\_BT table.

**Implementation Considerations.** There are various options for storing the tuples in a temporal relation. One is to store all the tuples in the same table, as shown in Figures 26.8 and 26.9. Another option is to create two tables: one for the currently valid information and the other for the rest of the tuples. For example, in the bitemporal EMP\_BT relation, tuples with  $uc$  for their Tet and *now* for their Vet would be in one relation, the *current table*, since they are the ones currently valid (that is, represent the current snapshot), and all other tuples would be in another relation. This allows the database administrator to have different access paths, such as indexes for each relation, and keeps the size of the current table reasonable. Another possibility is to create a third table for corrected tuples whose Tet is not  $uc$ .

Another option that is available is to *vertically partition* the attributes of the temporal relation into separate relations so that if a relation has many attributes, a whole new tuple version is created whenever any one of the attributes is updated. If the attributes are updated asynchronously, each new version may differ in only one of the attributes, thus needlessly repeating the other attribute values. If a separate relation is created to contain only the attributes that *always change synchronously*, with the primary key replicated in each relation, the database is said to be in **temporal normal form**. However, to combine the information, a variation of join known as **temporal intersection join** would be needed, which is generally expensive to implement.

It is important to note that bitemporal databases allow a complete record of changes. Even a record of corrections is possible. For example, it is possible that two tuple versions of the same employee may have the same valid time but different attribute values as long as their transaction times are disjoint. In this case, the tuple with the later transaction time is a **correction** of the other tuple version. Even incorrectly entered valid times may be corrected this way. The incorrect state of the database will still be available as a previous database state for querying purposes. A database that keeps such a complete record of changes and corrections is sometimes called an **append-only database**.

### 26.2.3 Incorporating Time in Object-Oriented Databases Using Attribute Versioning

The previous section discussed the **tuple versioning approach** to implementing temporal databases. In this approach, whenever one attribute value is changed, a whole new tuple version is created, even though all the other attribute values will

be identical to the previous tuple version. An alternative approach can be used in database systems that support **complex structured objects**, such as object databases (see Chapter 11) or object-relational systems. This approach is called **attribute versioning**.

In attribute versioning, a single complex object is used to store all the temporal changes of the object. Each attribute that changes over time is called a **time-varying attribute**, and it has its values versioned over time by adding temporal periods to the attribute. The temporal periods may represent valid time, transaction time, or bitemporal, depending on the application requirements. Attributes that do not change over time are called **non-time-varying** and are not associated with the temporal periods. To illustrate this, consider the example in Figure 26.10, which is an attribute-versioned valid time representation of EMPLOYEE

```

class TEMPORAL_SALARY
{
    attribute    Date           Valid_start_time;
    attribute    Date           Valid_end_time;
    attribute    float          Salary;
};

class TEMPORAL_DEPT
{
    attribute    Date           Valid_start_time;
    attribute    Date           Valid_end_time;
    attribute    DEPARTMENT_VT  Dept;
};

class TEMPORAL_SUPERVISOR
{
    attribute    Date           Valid_start_time;
    attribute    Date           Valid_end_time;
    attribute    EMPLOYEE_VT    Supervisor;
};

class TEMPORAL_LIFESPAN
{
    attribute    Date           Valid_ start time;
    attribute    Date           Valid end time;
};

class EMPLOYEE_VT
(
    extent EMPLOYEES )
{
    attribute    list<TEMPORAL_LIFESPAN>    lifespan;
    attribute    string                      Name;
    attribute    string                      Ssn;
    attribute    list<TEMPORAL_SALARY>       Sal_history;
    attribute    list<TEMPORAL_DEPT>         Dept_history;
    attribute    list <TEMPORAL_SUPERVISOR>  Supervisor_history;
};

```

**Figure 26.10**  
Possible ODL schema  
for a temporal valid  
time EMPLOYEE\_VT  
object class using  
attribute versioning.

using the object definition language (ODL) notation for object databases (see Chapter 11). Here, we assumed that name and Social Security number are non-time-varying attributes, whereas salary, department, and supervisor are time-varying attributes (they may change over time). Each time-varying attribute is represented as a list of tuples  $\langle \text{Valid\_start\_time}, \text{Valid\_end\_time}, \text{Value} \rangle$ , ordered by valid start time.

Whenever an attribute is changed in this model, the current attribute version is *closed* and a **new attribute version** for this attribute only is appended to the list. This allows attributes to change asynchronously. The current value for each attribute has *now* for its *Valid\_end\_time*. When using attribute versioning, it is useful to include a **lifespan temporal attribute** associated with the whole object whose value is one or more valid time periods that indicate the valid time of existence for the whole object. Logical deletion of the object is implemented by closing the lifespan. The constraint that any time period of an attribute within an object should be a subset of the object's lifespan should be enforced.

For bitemporal databases, each attribute version would have a tuple with five components:

$\langle \text{Valid\_start\_time}, \text{Valid\_end\_time}, \text{Trans\_start\_time}, \text{Trans\_end\_time}, \text{Value} \rangle$

The object lifespan would also include both valid and transaction time dimensions. Therefore, the full capabilities of bitemporal databases can be available with attribute versioning. Mechanisms similar to those discussed earlier for updating tuple versions can be applied to updating attribute versions.

## 26.2.4 Temporal Querying Constructs and the TSQL2 Language

So far, we have discussed how data models may be extended with temporal constructs. Now we give a brief overview of how query operations need to be extended for temporal querying. We will briefly discuss the TSQL2 language, which extends SQL for querying valid time and transaction time tables, as well as for querying of bitemporal relational tables.

In nontemporal relational databases, the typical selection conditions involve attribute conditions, and tuples that satisfy these conditions are selected from the set of *current tuples*. Following that, the attributes of interest to the query are specified by a *projection operation* (see Chapter 6). For example, in the query to retrieve the names of all employees working in department 5 whose salary is greater than 30000, the selection condition would be as follows:

$((\text{Salary} > 30000) \text{ AND } (\text{Dno} = 5))$

The projected attribute would be Name. In a temporal database, the conditions may involve time in addition to attributes. A **pure time condition** involves only time—for example, to select all employee tuple versions that were valid on a certain *time point*  $T$  or that were valid *during a certain time period*  $[T_1, T_2]$ . In this

case, the specified time period is compared with the valid time period of each tuple version  $[T.Vst, T.Vet]$ , and only those tuples that satisfy the condition are selected. In these operations, a period is considered to be equivalent to the set of time points from  $T_1$  to  $T_2$  inclusive, so the standard set comparison operations can be used. Additional operations, such as whether one time period ends *before* another starts, are also needed.<sup>21</sup>

Some of the more common operations used in queries are as follows:

$[T.Vst, T.Vet]$ <b>INCLUDES</b> $[T_1, T_2]$	Equivalent to $T_1 \geq T.Vst$ AND $T_2 \leq T.Vet$
$[T.Vst, T.Vet]$ <b>INCLUDED_IN</b> $[T_1, T_2]$	Equivalent to $T_1 \leq T.Vst$ AND $T_2 \geq T.Vet$
$[T.Vst, T.Vet]$ <b>OVERLAPS</b> $[T_1, T_2]$	Equivalent to $(T_1 \leq T.Vet$ AND $T_2 \geq T.Vst)$ <sup>22</sup>
$[T.Vst, T.Vet]$ <b>BEFORE</b> $[T_1, T_2]$	Equivalent to $T_1 \geq T.Vet$
$[T.Vst, T.Vet]$ <b>AFTER</b> $[T_1, T_2]$	Equivalent to $T_2 \leq T.Vst$
$[T.Vst, T.Vet]$ <b>MEETS_BEFORE</b> $[T_1, T_2]$	Equivalent to $T_1 = T.Vet + 1$ <sup>23</sup>
$[T.Vst, T.Vet]$ <b>MEETS_AFTER</b> $[T_1, T_2]$	Equivalent to $T_2 + 1 = T.Vst$

Additionally, operations are needed to manipulate time periods, such as computing the union or intersection of two time periods. The results of these operations may not themselves be periods, but rather **temporal elements**—a collection of one or more *disjoint* time periods such that no two time periods in a temporal element are directly adjacent. That is, for any two time periods  $[T_1, T_2]$  and  $[T_3, T_4]$  in a temporal element, the following three conditions must hold:

- $[T_1, T_2]$  intersection  $[T_3, T_4]$  is empty.
- $T_3$  is not the time point following  $T_2$  in the given granularity.
- $T_1$  is not the time point following  $T_4$  in the given granularity.

The latter conditions are necessary to ensure unique representations of temporal elements. If two time periods  $[T_1, T_2]$  and  $[T_3, T_4]$  are adjacent, they are combined into a single time period  $[T_1, T_4]$ . This is called **coalescing** of time periods. Coalescing also combines intersecting time periods.

To illustrate how pure time conditions can be used, suppose a user wants to select all employee versions that were valid at any point during 2002. The appropriate selection condition applied to the relation in Figure 26.8 would be

$[T.Vst, T.Vet]$  **OVERLAPS** [2002-01-01, 2002-12-31]

Typically, most temporal selections are applied to the valid time dimension. For a bitemporal database, one usually applies the conditions to the currently correct

<sup>21</sup>A complete set of operations, known as **Allen's algebra** (Allen, 1983), has been defined for comparing time periods.

<sup>22</sup>This operation returns true if the *intersection* of the two periods is not empty; it has also been called INTERSECTS\_WITH.

<sup>23</sup>Here, 1 refers to one time point in the specified granularity. The MEETS operations basically specify if one period starts immediately after another period ends.

tuples with *uc* as their transaction end times. However, if the query needs to be applied to a previous database state, an *AS\_OF T* clause is appended to the query, which means that the query is applied to the valid time tuples that were correct in the database at time *T*.

In addition to pure time conditions, other selections involve **attribute and time conditions**. For example, suppose we wish to retrieve all EMP\_VT tuple versions *T* for employees who worked in department 5 at any time during 2002. In this case, the condition is

$[T.Vst, T.Vet] \text{OVERLAPS } [2002-01-01, 2002-12-31] \text{ AND } (T.Dno = 5)$

Finally, we give a brief overview of the TSQL2 query language, which extends SQL with constructs for temporal databases. The main idea behind TSQL2 is to allow users to specify whether a relation is nontemporal (that is, a standard SQL relation) or temporal. The CREATE TABLE statement is extended with an *optional* AS clause to allow users to declare different temporal options. The following options are available:

- AS VALID STATE <GRANULARITY> (valid time relation with valid time period)
- AS VALID EVENT <GRANULARITY> (valid time relation with valid time point)
- AS TRANSACTION (transaction time relation with transaction time period)
- AS VALID STATE <GRANULARITY> AND TRANSACTION (bitemporal relation, valid time period)
- AS VALID EVENT <GRANULARITY> AND TRANSACTION (bitemporal relation, valid time point)

The keywords STATE and EVENT are used to specify whether a time *period* or time *point* is associated with the valid time dimension. In TSQL2, rather than have the user actually see how the temporal tables are implemented (as we discussed in the previous sections), the TSQL2 language adds query language constructs to specify various types of temporal selections, temporal projections, temporal aggregations, transformation among granularities, and many other concepts. The book by Snodgrass et al. (1995) describes the language.

### 26.2.5 Time Series Data

Time series data is used very often in financial, sales, and economics applications. They involve data values that are recorded according to a specific predefined sequence of time points. Therefore, they are a special type of **valid event data**, where the event's time points are predetermined according to a fixed calendar. Consider the example of closing daily stock prices of a particular company on the New York Stock Exchange. The granularity here is day, but the days that the stock market is open are known (non-holiday weekdays). Hence, it has been common to specify a computational procedure that calculates the particular **calendar** associated with a time series. Typical queries on

time series involve **temporal aggregation** over higher granularity intervals—for example, finding the average or maximum *weekly* closing stock price or the maximum and minimum *monthly* closing stock price from the *daily* information.

As another example, consider the daily sales dollar amount at each store of a chain of stores owned by a particular company. Again, typical temporal aggregates would be retrieving the weekly, monthly, or yearly sales from the daily sales information (using the sum aggregate function), or comparing same store monthly sales with previous monthly sales, and so on.

Because of the specialized nature of time series data and the lack of support for it in older DBMSs, it has been common to use specialized **time series management systems** rather than general-purpose DBMSs for managing such information. In such systems, it has been common to store time series values in sequential order in a file and apply specialized time series procedures to analyze the information. The problem with this approach is that the full power of high-level querying in languages such as SQL will not be available in such systems.

More recently, some commercial DBMS packages began offering time series extensions, such as the Oracle time cartridge and the time series data blade of Informix Universal Server. In addition, the TSQL2 language provides some support for time series in the form of event tables.

## 26.3 Spatial Database Concepts<sup>24</sup>

### 26.3.1 Introduction to Spatial Databases

Spatial databases incorporate functionality that provides support for databases that keep track of objects in a multidimensional space. For example, cartographic databases that store maps include two-dimensional spatial descriptions of their objects—from countries and states to rivers, cities, roads, seas, and so on. The systems that manage geographic data and related applications are known as **geographic information systems (GISs)**, and they are used in areas such as environmental applications, transportation systems, emergency response systems, and battle management. Other databases, such as meteorological databases for weather information, are three-dimensional, since temperatures and other meteorological information are related to three-dimensional spatial points. In general, a **spatial database** stores objects that have spatial characteristics that describe them and that have spatial relationships among them. The spatial relationships among the objects are important, and they are often needed when querying the database. Although a spatial database can in general refer to an  $n$ -dimensional space for any  $n$ , we will limit our discussion to two dimensions as an illustration.

A spatial database is optimized to store and query data related to objects in space, including points, lines and polygons. Satellite images are a prominent example of

---

<sup>24</sup>The contribution of Pranesh Parimala Ranganathan to this section is appreciated.

spatial data. Queries posed on these spatial data, where predicates for selection deal with spatial parameters, are called **spatial queries**. For example, “What are the names of all bookstores within five miles of the College of Computing building at Georgia Tech?” is a spatial query. Whereas typical databases process numeric and character data, additional functionality needs to be added for databases to process spatial data types. A query such as “List all the customers located within twenty miles of company headquarters” will require the processing of spatial data types typically outside the scope of standard relational algebra and may involve consulting an external geographic database that maps the company headquarters and each customer to a 2-D map based on their address. Effectively, each customer will be associated to a <latitude, longitude> position. A traditional B<sup>+</sup>-tree index based on customers’ zip codes or other nonspatial attributes cannot be used to process this query since traditional indexes are not capable of ordering multidimensional coordinate data. Therefore, there is a special need for databases tailored for handling spatial data and spatial queries.

Table 26.1 shows the common analytical operations involved in processing geographic or spatial data.<sup>25</sup> **Measurement operations** are used to measure some global properties of single objects (such as the area, the relative size of an object’s parts, compactness, or symmetry) and to measure the relative position of different objects in terms of distance and direction. **Spatial analysis** operations, which often use statistical techniques, are used to uncover *spatial relationships* within and among mapped data layers. An example would be to create a map—known as a *prediction map*—that identifies the locations of likely customers for particular products based on the historical sales and demographic information. **Flow analysis** operations help in determining the shortest path between two points and also the connectivity among nodes or regions in a graph. **Location analysis** aims to find if the given set of points and lines lie within a given polygon (location). The process involves generating a buffer around existing geographic features and then identifying or selecting features based on whether they fall inside or outside the boundary of the buffer. **Digital terrain analysis** is used to build three-dimensional models,

**Table 26.1** Common Types of Analysis for Spatial Data

Analysis Type	Type of Operations and Measurements
Measurements	Distance, perimeter, shape, adjacency, and direction
Spatial analysis/statistics	Pattern, autocorrelation, and indexes of similarity and topology using spatial and nonspatial data
Flow analysis	Connectivity and shortest path
Location analysis	Analysis of points and lines within a polygon
Terrain analysis	Slope/aspect, catchment area, drainage network
Search	Thematic search, search by region

<sup>25</sup>List of GIS analysis operations as proposed in Albrecht (1996).



where the topography of a geographical location can be represented with an  $x, y, z$  data model known as Digital Terrain (or Elevation) Model (DTM/DEM). The  $x$  and  $y$  dimensions of a DTM represent the horizontal plane, and  $z$  represents spot heights for the respective  $x, y$  coordinates. Such models can be used for analysis of environmental data or during the design of engineering projects that require terrain information. Spatial search allows a user to search for objects within a particular spatial region. For example, **thematic search** allows us to search for objects related to a particular theme or class, such as “Find all water bodies within 25 miles of Atlanta” where the class is *water*.

There are also **topological relationships** among spatial objects. These are often used in Boolean predicates to select objects based on their spatial relationships. For example, if a city boundary is represented as a polygon and freeways are represented as multilines, a condition such as “Find all freeways that go through Arlington, Texas” would involve an *intersects* operation, to determine which freeways (lines) intersect the city boundary (polygon).

### 26.3.2 Spatial Data Types and Models

This section briefly describes the common data types and models for storing spatial data. Spatial data comes in three basic forms. These forms have become a *de facto* standard due to their wide use in commercial systems.

- **Map data**<sup>26</sup> includes various geographic or spatial features of objects in a map, such as an object’s shape and the location of the object within the map. The three basic types of features are points, lines, and polygons (or areas). **Points** are used to represent spatial characteristics of objects whose locations correspond to a single 2-D coordinate ( $x, y$ , or longitude/latitude) in the scale of a particular application. Depending on the scale, some examples of point objects could be buildings, cellular towers, or stationary vehicles. Moving vehicles and other moving objects can be represented by a sequence of point locations that change over time. **Lines** represent objects having length, such as roads or rivers, whose spatial characteristics can be approximated by a sequence of connected lines. **Polygons** are used to represent spatial characteristics of objects that have a boundary, such as countries, states, lakes, or cities. Notice that some objects, such as buildings or cities, can be represented as either points or polygons, depending on the scale of detail.
- **Attribute data** is the descriptive data that GIS systems associate with **map features**. For example, suppose that a map contains features that represent counties within a U.S. state (such as Texas or Oregon). Attributes for each county feature (object) could include population, largest city/town, area in square miles, and so on. Other attribute data could be included for other features in the map, such as states, cities, congressional districts, census tracts, and so on.

---

<sup>26</sup>These types of geographic data are based on ESRI’s guide to GIS. See [www.gis.com/implementing\\_gis/data/data\\_types.html](http://www.gis.com/implementing_gis/data/data_types.html)



- **Image data** includes data such as satellite images and aerial photographs, which are typically created by cameras. Objects of interest, such as buildings and roads, can be identified and overlaid on these images. Images can also be attributes of map features. One can add images to other map features so that clicking on the feature would display the image. Aerial and satellite images are typical examples of raster data.

**Models of spatial information** are sometimes grouped into two broad categories: *field* and *object*. A spatial application (such as remote sensing or highway traffic control) is modeled using either a field- or an object-based model, depending on the requirements and the traditional choice of model for the application. **Field models** are often used to model spatial data that is continuous in nature, such as terrain elevation, temperature data, and soil variation characteristics, whereas **object models** have traditionally been used for applications such as transportation networks, land parcels, buildings, and other objects that possess both spatial and non-spatial attributes.

### 26.3.3 Spatial Operators and Spatial Queries

Spatial operators are used to capture all the relevant geometric properties of objects embedded in the physical space and the relations between them, as well as to perform spatial analysis. Operators are classified into three broad categories.

- **Topological operators.** Topological properties are invariant when topological transformations are applied. These properties do not change after transformations like rotation, translation, or scaling. Topological operators are hierarchically structured in several levels, where the base level offers operators the ability to check for detailed topological relations between regions with a broad boundary, and the higher levels offer more abstract operators that allow users to query uncertain spatial data independent of the underlying geometric data model. Examples include open (region), close (region), and inside (point, loop).
- **Projective operators.** Projective operators, such as *convex hull*, are used to express predicates about the concavity/convexity of objects as well as other spatial relations (for example, being inside the concavity of a given object).
- **Metric operators.** Metric operators provide a more specific description of the object's geometry. They are used to measure some global properties of single objects (such as the area, relative size of an object's parts, compactness, and symmetry), and to measure the relative position of different objects in terms of distance and direction. Examples include length (arc) and distance (point, point).

**Dynamic Spatial Operators.** The operations performed by the operators mentioned above are static, in the sense that the operands are not affected by the application of the operation. For example, calculating the length of the curve has no effect on the curve itself. **Dynamic operations** alter the objects upon which the operations act. The three fundamental dynamic operations are *create*, *destroy*, and

*update*. A representative example of dynamic operations would be updating a spatial object that can be subdivided into translate (shift position), rotate (change orientation), scale up or down, reflect (produce a mirror image), and shear (deform).

**Spatial Queries.** Spatial queries are requests for spatial data that require the use of spatial operations. The following categories illustrate three typical types of spatial queries:

- **Range queries.** Find all objects of a particular type that are within a given spatial area; for example, find all hospitals within the Metropolitan Atlanta city area. A variation of this query is to find all objects within a particular distance from a given location; for example, find all ambulances within a five mile radius of an accident location.
- **Nearest neighbor queries.** Finds an object of a particular type that is closest to a given location; for example, find the police car that is closest to the location of a crime. This can be generalized to find the  $k$  nearest neighbors, such as the 5 closest ambulances to an accident location.
- **Spatial joins or overlays.** Typically joins the objects of two types based on some spatial condition, such as the objects intersecting or overlapping spatially or being within a certain distance of one another. For example, find all townships located on a major highway between two cities or find all homes that are within two miles of a lake. The first example spatially joins *township* objects and *highway* object, and the second example spatially joins *lake* objects and *home* objects.

### 26.3.4 Spatial Data Indexing

A spatial index is used to organize objects into a set of buckets (which correspond to pages of secondary memory), so that objects in a particular spatial region can be easily located. Each bucket has a bucket region, a part of space containing all objects stored in the bucket. The bucket regions are usually rectangles; for point data structures, these regions are disjoint and they partition the space so that each point belongs to precisely one bucket. There are essentially two

1. Specialized indexing structures that allow efficient search for data objects based on spatial search operations are included in the database system. These indexing structures would play a similar role to that performed by  $B^+$ -tree indexes in traditional database systems. Examples of these indexing structures are *grid files* and *R-trees*. Special types of spatial indexes, known as *spatial join indexes*, can be used to speed up spatial join operations.
2. Instead of creating brand new indexing structures, the two-dimensional (2-D) spatial data is converted to single-dimensional (1-D) data, so that traditional indexing techniques ( $B^+$ -tree) can be used. The algorithms for converting from 2-D to 1-D are known as *space filling curves*. We will not discuss these methods in detail (see the Selected Bibliography for further references).

We give an overview of some of the spatial indexing techniques next.

**Grid Files.** We introduced grid files for indexing of data on multiple attributes in Chapter 18. They can also be used for indexing two-dimensional and higher  $n$ -dimensional spatial data. The **fixed-grid** method divides an  $n$ -dimensional hyperspace into equal size buckets. The data structure that implements the fixed grid is an  $n$ -dimensional array. The objects whose spatial locations lie within a cell (totally or partially) can be stored in a dynamic structure to handle overflows. This structure is useful for uniformly distributed data like satellite imagery. However, the fixed-grid structure is rigid, and its directory can be sparse and large.

**R-Trees.** The **R-tree** is a height-balanced tree, which is an extension of the  $B^+$ -tree for  $k$ -dimensions, where  $k > 1$ . For two dimensions (2-D), spatial objects are approximated in the  $R$ -tree by their **minimum bounding rectangle (MBR)**, which is the smallest rectangle, with sides parallel to the coordinate system ( $x$  and  $y$ ) axis, that contains the object.  $R$ -trees are characterized by the following properties, which are similar to the properties for  $B^+$ -trees (see Section 18.3) but are adapted to 2-D spatial objects. As in Section 18.3, we use  $M$  to indicate the maximum number of entries that can fit in an  $R$ -tree node.

1. The structure of each index entry (or index record) in a leaf node is  $(I, \text{object-identifier})$ , where  $I$  is the MBR for the spatial object whose identifier is *object-identifier*.
2. Every node except the root node must be at least half full. Thus, a leaf node that is not the root should contain  $m$  entries  $(I, \text{object-identifier})$  where  $M/2 \leq m \leq M$ . Similarly, a non-leaf node that is not the root should contain  $m$  entries  $(I, \text{child-pointer})$  where  $M/2 \leq m \leq M$ , and  $I$  is the MBR that contains the union of all the rectangles in the node pointed at by *child-pointer*.
3. All leaf nodes are at the same level, and the root node should have at least two pointers unless it is a leaf node.
4. All MBRs have their sides parallel to the axes of the global coordinate system.

Other spatial storage structures include quadtrees and their variations. **Quadtrees** generally divide each space or subspace into equally sized areas and proceed with the subdivisions of each subspace to identify the positions of various objects. Recently, many newer spatial access structures have been proposed, and this remains an active research area.

**Spatial Join Index.** A spatial join index precomputes a spatial join operation and stores the pointers to the related object in an index structure. Join indexes improve the performance of recurring join queries over tables that have low update rates. Spatial join conditions are used to answer queries such as “Create a list of highway-river combinations that cross.” The spatial join is used to identify and retrieve these pairs of objects that satisfy the *cross* spatial relationship. Because computing the results of spatial relationships is generally time consuming, the result can be computed once and stored in a table that has the pairs of object identifiers (or tuple ids) that satisfy the spatial relationship, which is essentially the join index.

A join index can be described by a bipartite graph  $G = (V_1, V_2, E)$ , where  $V_1$  contains the tuple ids of relation  $R$  and  $V_2$  contains the tuple ids of relation  $S$ . Edge set contains an edge  $(v_r, v_s)$  for  $v_r$  in  $R$  and  $v_s$  in  $S$ , if there is a tuple corresponding to  $(v_r, v_s)$  in the join index. The bipartite graph models all of the related tuples as connected vertices in the graphs. Spatial join indexes are used in operations (see Section 26.3.3) that involve computation of relationships among spatial objects.

### 26.3.5 Spatial Data Mining

Spatial data tends to be highly correlated. For example, people with similar characteristics, occupations, and backgrounds tend to cluster together in the same neighborhoods. The three major spatial data mining techniques are spatial classification, spatial association, and spatial clustering.

- **Spatial classification.** The goal of classification is to estimate the value of an attribute of a relation based on the value of the relation's other attributes. An example of the spatial classification problem is determining the locations of nests in a wetland based on the value of other attributes (for example, vegetation durability and water depth); it is also called the *location prediction problem*. Similarly, where to expect hotspots in crime activity is also a location prediction problem.
- **Spatial association. Spatial association rules** are defined in terms of spatial predicates rather than items. A spatial association rule is of the form

$$P_1 \wedge P_2 \wedge \dots \wedge P_n \Rightarrow Q_1 \wedge Q_2 \wedge \dots \wedge Q_m$$

where at least one of the  $P_i$ 's or  $Q_j$ 's is a spatial predicate. For example, the rule

$$\text{is\_a}(x, \text{country}) \wedge \text{touches}(x, \text{Mediterranean}) \Rightarrow \text{is\_a}(x, \text{wine-exporter})$$

(that is, a country that is adjacent to the Mediterranean Sea is typically a wine exporter) is an example of an association rule, which will have a certain support  $s$  and confidence  $c$ .<sup>27</sup>

**Spatial colocation rules** attempt to generalize association rules to point to collection data sets that are indexed by space. There are several crucial differences between spatial and nonspatial associations, including the following:

1. The notion of a transaction is absent in spatial situations, since data is embedded in continuous space. Partitioning space into transactions would lead to an overestimate or an underestimate of interest measures, for example, support or confidence.
2. Size of item sets in spatial databases is small, that is, there are many fewer items in the item set in a spatial situation than in a nonspatial situation.

<sup>27</sup>Concepts of support and confidence for association rules are discussed as part of data mining in Section 28.2.

In most instances, spatial items are a discrete version of continuous variables. For example, in the United States income regions may be defined as regions where the mean yearly income is within certain ranges, such as, below \$40,000, from \$40,000 to \$100,000, and above \$100,000.

- **Spatial clustering** attempts to group database objects so that the most similar objects are in the same cluster, and objects in different clusters are as dissimilar as possible. One application of spatial clustering is to group together seismic events in order to determine earthquake faults. An example of a spatial clustering algorithm is **density-based clustering**, which tries to find clusters based on the density of data points in a region. These algorithms treat clusters as dense regions of objects in the data space. Two variations of these algorithms are density-based spatial clustering of applications with noise (DBSCAN)<sup>28</sup> and density-based clustering (DENCLUE).<sup>29</sup> DBSCAN is a density-based clustering algorithm because it finds a number of clusters starting from the estimated density distribution of corresponding nodes.

### 26.3.6 Applications of Spatial Data

Spatial data management is useful in many disciplines, including geography, remote sensing, urban planning, and natural resource management. Spatial database management is playing an important role in the solution of challenging scientific problems such as global climate change and genomics. Due to the spatial nature of genome data, GIS and spatial database management systems have a large role to play in the area of bioinformatics. Some of the typical applications include pattern recognition (for example, to check if the topology of a particular gene in the genome is found in any other sequence feature map in the database), genome browser development, and visualization maps. Another important application area of spatial data mining is the spatial outlier detection. A **spatial outlier** is a spatially referenced object whose nonspatial attribute values are significantly different from those of other spatially referenced objects in its spatial neighborhood. For example, if a neighborhood of older houses has just one brand-new house, that house would be an outlier based on the nonspatial attribute 'house\_age'. Detecting spatial outliers is useful in many applications of geographic information systems and spatial databases. These application domains include transportation, ecology, public safety, public health, climatology, and location-based services.

## 26.4 Multimedia Database Concepts

**Multimedia databases** provide features that allow users to store and query different types of multimedia information, which includes *images* (such as photos or drawings), *video clips* (such as movies, newsreels, or home videos), *audio clips*

<sup>28</sup>DBSCAN was proposed by Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu (1996).

<sup>29</sup>DENCLUE was proposed by Hinnenberg and Gabriel (2007).

(such as songs, phone messages, or speeches), and *documents* (such as books or articles). The main types of database queries that are needed involve locating multimedia sources that contain certain objects of interest. For example, one may want to locate all video clips in a video database that include a certain person, say Michael Jackson. One may also want to retrieve video clips based on certain activities included in them, such as video clips where a soccer goal is scored by a certain player or team.

The above types of queries are referred to as **content-based retrieval**, because the multimedia source is being retrieved based on its containing certain objects or activities. Hence, a multimedia database must use some model to organize and index the multimedia sources based on their contents. *Identifying the contents* of multimedia sources is a difficult and time-consuming task. There are two main approaches. The first is based on **automatic analysis** of the multimedia sources to identify certain mathematical characteristics of their contents. This approach uses different techniques depending on the type of multimedia source (image, video, audio, or text). The second approach depends on **manual identification** of the objects and activities of interest in each multimedia source and on using this information to index the sources. This approach can be applied to all multimedia sources, but it requires a manual preprocessing phase in which a person must scan each multimedia source to identify and catalog the objects and activities it contains so that they can be used to index the sources.

In the first part of this section, we will briefly discuss some of the characteristics of each type of multimedia source—images, video, audio, and text/documents. Then we will discuss approaches for automatic analysis of images followed by the problem of object recognition in images. We end this section with some remarks on analyzing audio sources.

An **image** is typically stored either in raw form as a set of pixel or cell values, or in compressed form to save space. The image *shape descriptor* describes the geometric shape of the raw image, which is typically a rectangle of **cells** of a certain width and height. Hence, each image can be represented by an  $m$  by  $n$  grid of cells. Each cell contains a pixel value that describes the cell content. In black-and-white images, pixels can be one bit. In grayscale or color images, a pixel is multiple bits. Because images may require large amounts of space, they are often stored in compressed form. Compression standards, such as GIF, JPEG, or MPEG, use various mathematical transformations to reduce the number of cells stored but still maintain the main image characteristics. Applicable mathematical transforms include discrete Fourier transform (DFT), discrete cosine transform (DCT), and wavelet transforms.

To identify objects of interest in an image, the image is typically divided into homogeneous segments using a *homogeneity predicate*. For example, in a color image, adjacent cells that have similar pixel values are grouped into a segment. The homogeneity predicate defines conditions for automatically grouping those cells. Segmentation and compression can hence identify the main characteristics of an image.

A typical image database query would be to find images in the database that are similar to a given image. The given image could be an isolated segment that contains, say, a pattern of interest, and the query is to locate other images that contain that same pattern. There are two main techniques for this type of search. The first approach uses a **distance function** to compare the given image with the stored images and their segments. If the distance value returned is small, the probability of a match is high. Indexes can be created to group stored images that are close in the distance metric so as to limit the search space. The second approach, called the **transformation approach**, measures image similarity by having a small number of transformations that can change one image's cells to match the other image. Transformations include rotations, translations, and scaling. Although the transformation approach is more general, it is also more time-consuming and difficult.

A **video source** is typically represented as a sequence of frames, where each frame is a still image. However, rather than identifying the objects and activities in every individual frame, the video is divided into **video segments**, where each segment comprises a sequence of contiguous frames that includes the same objects/activities. Each segment is identified by its starting and ending frames. The objects and activities identified in each video segment can be used to index the segments. An indexing technique called *frame segment trees* has been proposed for video indexing. The index includes both objects, such as persons, houses, and cars, as well as activities, such as a person *delivering* a speech or two people *talking*. Videos are also often compressed using standards such as MPEG.

**Audio sources** include stored recorded messages, such as speeches, class presentations, or even surveillance recordings of phone messages or conversations by law enforcement. Here, discrete transforms can be used to identify the main characteristics of a certain person's voice in order to have similarity-based indexing and retrieval. We will briefly comment on their analysis in Section 26.4.4.

A **text/document source** is basically the full text of some article, book, or magazine. These sources are typically indexed by identifying the keywords that appear in the text and their relative frequencies. However, filler words or common words called **stopwords** are eliminated from the process. Because there can be many keywords when attempting to index a collection of documents, techniques have been developed to reduce the number of keywords to those that are most relevant to the collection. A dimensionality reduction technique called *singular value decomposition* (SVD), which is based on matrix transformations, can be used for this purpose. An indexing technique called *telescoping vector trees* (TV-trees) can then be used to group similar documents. Chapter 27 discusses document processing in detail.

### 26.4.1 Automatic Analysis of Images

Analysis of multimedia sources is critical to support any type of query or search interface. We need to represent multimedia source data such as images in terms of features that would enable us to define similarity. The work done so far in this area uses low-level visual features such as color, texture, and shape, which are directly



related to the perceptual aspects of image content. These features are easy to extract and represent, and it is convenient to design similarity measures based on their statistical properties.

**Color** is one of the most widely used visual features in content-based image retrieval since it does not depend upon image size or orientation. Retrieval based on color similarity is mainly done by computing a color histogram for each image that identifies the proportion of pixels within an image for the three color channels (red, green, blue—**RGB**). However, RGB representation is affected by the orientation of the object with respect to illumination and camera direction. Therefore, current image retrieval techniques compute color histograms using competing invariant representations such as **HSV** (hue, saturation, value). HSV describes colors as points in a cylinder whose central axis ranges from black at the bottom to white at the top with neutral colors between them. The angle around the axis corresponds to the hue, the distance from the axis corresponds to the saturation, and the distance along the axis corresponds to the value (brightness).

**Texture** refers to the patterns in an image that present the properties of homogeneity that do not result from the presence of a single color or intensity value. Examples of texture classes are rough and silky. Examples of textures that can be identified include pressed calf leather, straw matting, cotton canvas, and so on. Just as pictures are represented by arrays of pixels (picture elements), textures are represented by **arrays of texels** (texture elements). These textures are then placed into a number of sets, depending on how many textures are identified in the image. These sets not only contain the texture definition but also indicate where in the image the texture is located. Texture identification is primarily done by modeling it as a two-dimensional, gray-level variation. The relative brightness of pairs of pixels is computed to estimate the degree of contrast, regularity, coarseness, and directionality.

**Shape** refers to the shape of a region within an image. It is generally determined by applying segmentation or edge detection to an image. **Segmentation** is a region-based approach that uses an entire region (sets of pixels), whereas **edge detection** is a boundary-based approach that uses only the outer boundary characteristics of entities. Shape representation is typically required to be invariant to translation, rotation, and scaling. Some well-known methods for shape representation include Fourier descriptors and moment invariants.

## 26.4.2 Object Recognition in Images

**Object recognition** is the task of identifying real-world objects in an image or a video sequence. The system must be able to identify the object even when the images of the object vary in viewpoints, size, scale, or even when they are rotated or translated. Some approaches have been developed to divide the original image into regions based on similarity of contiguous pixels. Thus, in a given image showing a tiger in the jungle, a tiger subimage may be detected against the background of the jungle, and when compared with a set of training images, it may be tagged as a tiger.



The representation of the multimedia object in an object model is extremely important. One approach is to divide the image into homogeneous segments using a homogeneous predicate. For example, in a colored image, adjacent cells that have similar pixel values are grouped into a segment. The homogeneity predicate defines conditions for automatically grouping those cells. Segmentation and compression can hence identify the main characteristics of an image. Another approach finds measurements of the object that are invariant to transformations. It is impossible to keep a database of examples of all the different transformations of an image. To deal with this, object recognition approaches find interesting points (or features) in an image that are invariant to transformations.

An important contribution to this field was made by Lowe,<sup>30</sup> who used scale-invariant features from images to perform reliable object recognition. This approach is called **scale-invariant feature transform (SIFT)**. The SIFT features are invariant to image scaling and rotation, and partially invariant to change in illumination and 3D camera viewpoint. They are well localized in both the spatial and frequency domains, reducing the probability of disruption by occlusion, clutter, or noise. In addition, the features are highly distinctive, which allows a single feature to be correctly matched with high probability against a large database of features, providing a basis for object and scene recognition.

For image matching and recognition, SIFT features (also known as *keypoint features*) are first extracted from a set of reference images and stored in a database. Object recognition is then performed by comparing each feature from the new image with the features stored in the database and finding candidate matching features based on the Euclidean distance of their feature vectors. Since the keypoint features are highly distinctive, a single feature can be correctly matched with good probability in a large database of features.

In addition to SIFT, there are a number of competing methods available for object recognition under clutter or partial occlusion. For example, **RIFT**, a rotation invariant generalization of SIFT, identifies groups of local affine regions (image features having a characteristic appearance and elliptical shape) that remain approximately affinely rigid across a range of views of an object, and across multiple instances of the same object class.

### 26.4.3 Semantic Tagging of Images

The notion of implicit tagging is an important one for image recognition and comparison. Multiple tags may attach to an image or a subimage: for instance, in the example we referred to above, tags such as “tiger,” “jungle,” “green,” and “stripes” may be associated with that image. Most image search techniques retrieve images based on user-supplied tags that are often not very accurate or comprehensive. To improve search quality, a number of recent systems aim at automated generation of these image tags. In case of multimedia data, most of its semantics is present in its

---

<sup>30</sup>See Lowe (2004), “Distinctive Image Features from Scale-Invariant Keypoints.”

content. These systems use image-processing and statistical-modeling techniques to analyze image content to generate accurate annotation tags that can then be used to retrieve images by content. Since different annotation schemes will use different vocabularies to annotate images, the quality of image retrieval will be poor. To solve this problem, recent research techniques have proposed the use of concept hierarchies, taxonomies, or ontologies using **OWL (Web Ontology Language)**, in which terms and their relationships are clearly defined. These can be used to infer higher-level concepts based on tags. Concepts like “sky” and “grass” may be further divided into “clear sky” and “cloudy sky” or “dry grass” and “green grass” in such a taxonomy. These approaches generally come under semantic tagging and can be used in conjunction with the above feature-analysis and object-identification strategies.

#### 26.4.4 Analysis of Audio Data Sources

Audio sources are broadly classified into speech, music, and other audio data. Each of these is significantly different from the others; hence different types of audio data are treated differently. Audio data must be digitized before it can be processed and stored. Indexing and retrieval of audio data is arguably the toughest among all types of media, because like video, it is continuous in time and does not have easily measurable characteristics such as text. Clarity of sound recordings is easy to perceive humanly but is hard to quantify for machine learning. Interestingly, speech data often uses speech recognition techniques to aid the actual audio content, as this can make indexing this data a lot easier and more accurate. This is sometimes referred to as *text-based indexing of audio data*. The speech metadata is typically content dependent, in that the metadata is generated from the audio content; for example, the length of the speech, the number of speakers, and so on. However, some of the metadata might be independent of the actual content, such as the length of the speech and the format in which the data is stored. Music indexing, on the other hand, is done based on the statistical analysis of the audio signal, also known as *content-based indexing*. Content-based indexing often makes use of the key features of sound: intensity, pitch, timbre, and rhythm. It is possible to compare different pieces of audio data and retrieve information from them based on the calculation of certain features, as well as application of certain transforms.

## 26.5 Introduction to Deductive Databases

### 26.5.1 Overview of Deductive Databases

In a deductive database system we typically specify rules through a **declarative language**—a language in which we specify what to achieve rather than how to achieve it. An **inference engine** (or **deduction mechanism**) within the system can deduce new facts from the database by interpreting these rules. The model used for deductive databases is closely related to the relational data model, and particularly to the domain relational calculus formalism (see Section 6.6). It is also related to the field of **logic programming** and the **Prolog** language. The deductive database work

based on logic has used Prolog as a starting point. A variation of Prolog called **Datalog** is used to define rules declaratively in conjunction with an existing set of relations, which are themselves treated as literals in the language. Although the language structure of Datalog resembles that of Prolog, its operational semantics—that is, how a Datalog program is executed—is still different.

A deductive database uses two main types of specifications: facts and rules. **Facts** are specified in a manner similar to the way relations are specified, except that it is not necessary to include the attribute names. Recall that a tuple in a relation describes some real-world fact whose meaning is partly determined by the attribute names. In a deductive database, the meaning of an attribute value in a tuple is determined solely by its *position* within the tuple. **Rules** are somewhat similar to relational views. They specify virtual relations that are not actually stored but that can be formed from the facts by applying inference mechanisms based on the rule specifications. The main difference between rules and views is that rules may involve recursion and hence may yield virtual relations that cannot be defined in terms of basic relational views.

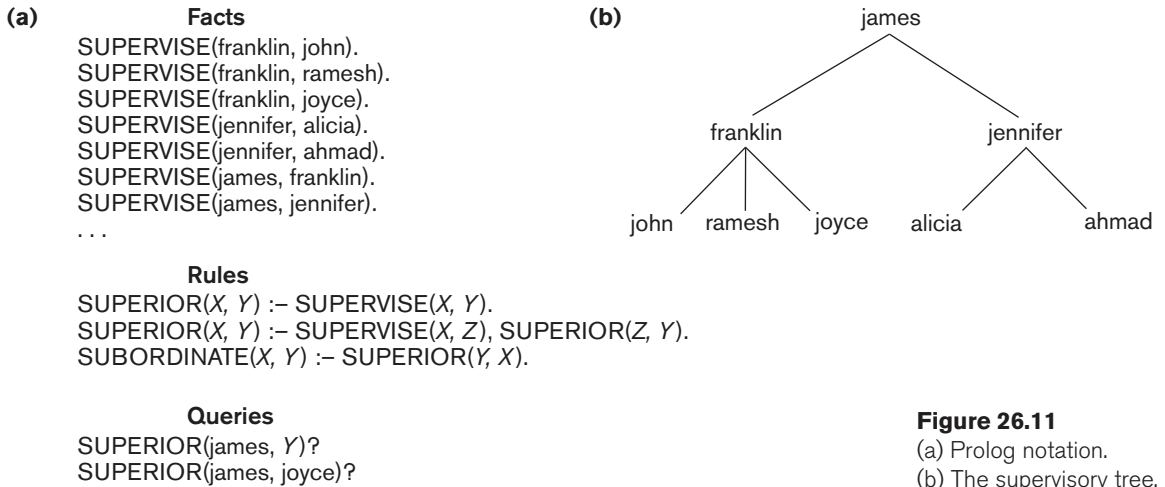
The evaluation of Prolog programs is based on a technique called *backward chaining*, which involves a top-down evaluation of goals. In the deductive databases that use Datalog, attention has been devoted to handling large volumes of data stored in a relational database. Hence, evaluation techniques have been devised that resemble those for a bottom-up evaluation. Prolog suffers from the limitation that the order of specification of facts and rules is significant in evaluation; moreover, the order of literals (defined in Section 26.5.3) within a rule is significant. The execution techniques for Datalog programs attempt to circumvent these problems.

## 26.5.2 Prolog/Datalog Notation

The notation used in Prolog/Datalog is based on providing predicates with unique names. A **predicate** has an implicit meaning, which is suggested by the predicate name, and a fixed number of **arguments**. If the arguments are all constant values, the predicate simply states that a certain fact is true. If, on the other hand, the predicate has variables as arguments, it is either considered as a query or as part of a rule or constraint. In our discussion, we adopt the Prolog convention that all **constant values** in a predicate are either *numeric* or *character strings*; they are represented as identifiers (or names) that start with a *lowercase letter*, whereas **variable names** always start with an *uppercase letter*.

Consider the example shown in Figure 26.11, which is based on the relational database in Figure 3.6, but in a much simplified form. There are three predicate names: *supervise*, *superior*, and *subordinate*. The SUPERVISE predicate is defined via a set of facts, each of which has two arguments: a supervisor name, followed by the name of a *direct* supervisee (subordinate) of that supervisor. These facts correspond to the actual data that is stored in the database, and they can be considered as constituting a set of tuples in a relation SUPERVISE with two attributes whose schema is

SUPERVISE(Supervisor, Supervisee)



**Figure 26.11**  
(a) Prolog notation.  
(b) The supervisory tree.

Thus,  $\text{SUPERVISE}(X, Y)$  states the fact that  $X$  *supervises*  $Y$ . Notice the omission of the attribute names in the Prolog notation. Attribute names are only represented by virtue of the position of each argument in a predicate: the first argument represents the supervisor, and the second argument represents a direct subordinate.

The other two predicate names are defined by rules. The main contributions of deductive databases are the ability to specify recursive rules and to provide a framework for inferring new information based on the specified rules. A rule is of the form **head**  $\text{:-}$  **body**, where  $\text{:-}$  is read as *if and only if*. A rule usually has a single **predicate** to the left of the  $\text{:-}$  symbol—called the **head** or **left-hand side (LHS)** or **conclusion** of the rule—and *one or more* **predicates** to the right of the  $\text{:-}$  symbol—called the **body** or **right-hand side (RHS)** or **premise(s)** of the rule. A predicate with constants as arguments is said to be **ground**; we also refer to it as an **instantiated predicate**. The arguments of the predicates that appear in a rule typically include a number of variable symbols, although predicates can also contain constants as arguments. A rule specifies that, if a particular assignment or **binding** of constant values to the variables in the body (RHS predicates) makes *all* the RHS predicates **true**, it also makes the head (LHS predicate) true by using the same assignment of constant values to variables. Hence, a rule provides us with a way of generating new facts that are instantiations of the head of the rule. These new facts are based on facts that already exist, corresponding to the instantiations (or bindings) of predicates in the body of the rule. Notice that by listing multiple predicates in the body of a rule we implicitly apply the **logical AND** operator to these predicates. Hence, the commas between the RHS predicates may be read as meaning *and*.

Consider the definition of the predicate  $\text{SUPERIOR}$  in Figure 26.11, whose first argument is an employee name and whose second argument is an employee who is either a *direct* or an *indirect* subordinate of the first employee. By *indirect subordinate*, we

mean the subordinate of some subordinate down to any number of levels. Thus  $\text{SUPERIOR}(X, Y)$  stands for the fact that  $X$  is a *superior* of  $Y$  through direct or indirect supervision. We can write two rules that together specify the meaning of the new predicate. The first rule under Rules in the figure states that for every value of  $X$  and  $Y$ , if  $\text{SUPERVISE}(X, Y)$ —the rule body—is true, then  $\text{SUPERIOR}(X, Y)$ —the rule head—is also true, since  $Y$  would be a direct subordinate of  $X$  (at one level down). This rule can be used to generate all direct superior/subordinate relationships from the facts that define the  $\text{SUPERVISE}$  predicate. The second recursive rule states that if  $\text{SUPERVISE}(X, Z)$  and  $\text{SUPERIOR}(Z, Y)$  are *both* true, then  $\text{SUPERIOR}(X, Y)$  is also true. This is an example of a **recursive rule**, where one of the rule body predicates in the RHS is the same as the rule head predicate in the LHS. In general, the rule body defines a number of premises such that if they are all true, we can deduce that the conclusion in the rule head is also true. Notice that if we have two (or more) rules with the same head (LHS predicate), it is equivalent to saying that the predicate is true (that is, that it can be instantiated) if *either one* of the bodies is true; hence, it is equivalent to a **logical OR** operation. For example, if we have two rules  $X :- Y$  and  $X :- Z$ , they are equivalent to a rule  $X :- Y \text{ OR } Z$ . The latter form is not used in deductive systems, however, because it is not in the standard form of rule, called a *Horn clause*, as we discuss in Section 26.5.4.

A Prolog system contains a number of **built-in** predicates that the system can interpret directly. These typically include the equality comparison operator  $= (X, Y)$ , which returns true if  $X$  and  $Y$  are identical and can also be written as  $X = Y$  by using the standard infix notation.<sup>31</sup> Other comparison operators for numbers, such as  $<$ ,  $<=$ ,  $>$ , and  $>=$ , can be treated as binary predicates. Arithmetic functions such as  $+$ ,  $-$ ,  $*$ , and  $/$  can be used as arguments in predicates in Prolog. In contrast, Datalog (in its basic form) does *not* allow functions such as arithmetic operations as arguments; indeed, this is one of the main differences between Prolog and Datalog. However, extensions to Datalog have been proposed that do include functions.

A **query** typically involves a predicate symbol with some variable arguments, and its meaning (or *answer*) is to deduce all the different constant combinations that, when **bound** (assigned) to the variables, can make the predicate true. For example, the first query in Figure 26.11 requests the names of all subordinates of *james* at any level. A different type of query, which has only constant symbols as arguments, returns either a true or a false result, depending on whether the arguments provided can be deduced from the facts and rules. For example, the second query in Figure 26.11 returns true, since  $\text{SUPERIOR}(\text{james}, \text{joyce})$  can be deduced.

### 26.5.3 Datalog Notation

In Datalog, as in other logic-based languages, a program is built from basic objects called **atomic formulas**. It is customary to define the syntax of logic-based languages by describing the syntax of atomic formulas and identifying how they can be combined to form a program. In Datalog, atomic formulas are **literals** of the form

<sup>31</sup>A Prolog system typically has a number of different equality predicates that have different interpretations.

$p(a_1, a_2, \dots, a_n)$ , where  $p$  is the predicate name and  $n$  is the number of arguments for predicate  $p$ . Different predicate symbols can have different numbers of arguments, and the number of arguments  $n$  of predicate  $p$  is sometimes called the **arity** or **degree** of  $p$ . The arguments can be either constant values or variable names. As mentioned earlier, we use the convention that constant values either are numeric or start with a *lowercase* character, whereas variable names always start with an *uppercase* character.

A number of **built-in predicates** are included in Datalog and can also be used to construct atomic formulas. The built-in predicates are of two main types: the binary comparison predicates  $<$  (less),  $<=$  (less\_or\_equal),  $>$  (greater), and  $>=$  (greater\_or\_equal) over ordered domains; and the comparison predicates  $=$  (equal) and  $\neq$  (not\_equal) over ordered or unordered domains. These can be used as binary predicates with the same functional syntax as other predicates—for example, by writing `less(X, 3)`—or they can be specified by using the customary infix notation  $X < 3$ . Note that because the domains of these predicates are potentially infinite, they should be used with care in rule definitions. For example, the predicate `greater(X, 3)`, if used alone, generates an infinite set of values for  $X$  that satisfy the predicate (all integer numbers greater than 3).

A **literal** is either an atomic formula as defined earlier—called a **positive literal**—or an atomic formula preceded by **not**. The latter is a negated atomic formula, called a **negative literal**. Datalog programs can be considered to be a *subset* of the predicate calculus formulas, which are somewhat similar to the formulas of the domain relational calculus (see Section 6.7). In Datalog, however, these formulas are first converted into what is known as **clausal form** before they are expressed in Datalog, and only formulas given in a restricted clausal form, called *Horn clauses*,<sup>32</sup> can be used in Datalog.

### 26.5.4 Clausal Form and Horn Clauses

Recall from Section 6.6 that a formula in the relational calculus is a condition that includes predicates called *atoms* (based on relation names). Additionally, a formula can have quantifiers—namely, the *universal quantifier* (for all) and the *existential quantifier* (there exists). In clausal form, a formula must be transformed into another formula with the following characteristics:

- All variables in the formula are universally quantified. Hence, it is not necessary to include the universal quantifiers (for all) explicitly; the quantifiers are removed, and all variables in the formula are *implicitly* quantified by the universal quantifier.
- In clausal form, the formula is made up of a number of clauses, where each **clause** is composed of a number of *literals* connected by OR logical connectives only. Hence, each clause is a *disjunction* of literals.
- The *clauses themselves* are connected by AND logical connectives only, to form a formula. Hence, the **clausal form of a formula** is a *conjunction* of clauses.

---

<sup>32</sup>Named after the mathematician Alfred Horn.

It can be shown that *any formula can be converted into clausal form*. For our purposes, we are mainly interested in the form of the individual clauses, each of which is a disjunction of literals. Recall that literals can be positive literals or negative literals. Consider a clause of the form:

$$\text{NOT}(P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \dots \text{ OR } \text{NOT}(P_n) \text{ OR } Q_1 \text{ OR } Q_2 \text{ OR } \dots \text{ OR } Q_m \quad (1)$$

This clause has  $n$  negative literals and  $m$  positive literals. Such a clause can be transformed into the following equivalent logical formula:

$$P_1 \text{ AND } P_2 \text{ AND } \dots \text{ AND } P_n \Rightarrow Q_1 \text{ OR } Q_2 \text{ OR } \dots \text{ OR } Q_m \quad (2)$$

where  $\Rightarrow$  is the **implies** symbol. The formulas (1) and (2) are equivalent, meaning that their truth values are always the same. This is the case because if all the  $P_i$  literals ( $i = 1, 2, \dots, n$ ) are true, the formula (2) is true only if at least one of the  $Q_i$ 's is true, which is the meaning of the  $\Rightarrow$  (implies) symbol. For formula (1), if all the  $P_i$  literals ( $i = 1, 2, \dots, n$ ) are true, their negations are all false; so in this case formula (1) is true only if at least one of the  $Q_i$ 's is true. In Datalog, rules are expressed as a restricted form of clauses called **Horn clauses**, in which a clause can contain *at most one* positive literal. Hence, a Horn clause is either of the form

$$\text{NOT}(P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \dots \text{ OR } \text{NOT}(P_n) \text{ OR } Q \quad (3)$$

or of the form

$$\text{NOT}(P_1) \text{ OR } \text{NOT}(P_2) \text{ OR } \dots \text{ OR } \text{NOT}(P_n) \quad (4)$$

The Horn clause in (3) can be transformed into the clause

$$P_1 \text{ AND } P_2 \text{ AND } \dots \text{ AND } P_n \Rightarrow Q \quad (5)$$

which is written in Datalog as the following rule:

$$Q :- P_1, P_2, \dots, P_n. \quad (6)$$

The Horn clause in (4) can be transformed into

$$P_1 \text{ AND } P_2 \text{ AND } \dots \text{ AND } P_n \Rightarrow \quad (7)$$

which is written in Datalog as follows:

$$P_1, P_2, \dots, P_n. \quad (8)$$

A **Datalog rule**, as in (6), is hence a Horn clause, and its meaning, based on formula (5), is that if the predicates  $P_1 \text{ AND } P_2 \text{ AND } \dots \text{ AND } P_n$  are all true for a particular binding to their variable arguments, then  $Q$  is also true and can hence be inferred. The Datalog expression (8) can be considered as an integrity constraint, where all the predicates must be true to satisfy the query.

In general, a **query in Datalog** consists of two components:

- A Datalog program, which is a finite set of rules
- A literal  $P(X_1, X_2, \dots, X_n)$ , where each  $X_i$  is a variable or a constant

A Prolog or Datalog system has an internal **inference engine** that can be used to process and compute the results of such queries. Prolog inference engines typically



return one result to the query (that is, one set of values for the variables in the query) at a time and must be prompted to return additional results. On the contrary, Datalog returns results set-at-a-time.

### 26.5.5 Interpretations of Rules

There are two main alternatives for interpreting the theoretical meaning of rules: *proof-theoretic* and *model-theoretic*. In practical systems, the inference mechanism within a system defines the exact interpretation, which may not coincide with either of the two theoretical interpretations. The inference mechanism is a computational procedure and hence provides a computational interpretation of the meaning of rules. In this section, first we discuss the two theoretical interpretations. Then we briefly discuss inference mechanisms as a way of defining the meaning of rules.

In the **proof-theoretic** interpretation of rules, we consider the facts and rules to be true statements, or **axioms**. **Ground axioms** contain no variables. The facts are ground axioms that are given to be true. Rules are called **deductive axioms**, since they can be used to deduce new facts. The deductive axioms can be used to construct proofs that derive new facts from existing facts. For example, Figure 26.12 shows how to prove the fact SUPERIOR(james, ahmad) from the rules and facts given in Figure 26.11. The proof-theoretic interpretation gives us a procedural or computational approach for computing an answer to the Datalog query. The process of proving whether a certain fact (theorem) holds is known as **theorem proving**.

The second type of interpretation is called the **model-theoretic** interpretation. Here, given a finite or an infinite domain of constant values,<sup>33</sup> we assign to a predicate every possible combination of values as arguments. We must then determine whether the predicate is true or false. In general, it is sufficient to specify the combinations of arguments that make the predicate true, and to state that all other combinations make the predicate false. If this is done for every predicate, it is called an **interpretation** of the set of predicates. For example, consider the interpretation shown in Figure 26.13 for the predicates SUPERVISE and SUPERIOR. This interpretation assigns a truth value (true or false) to every possible combination of argument values (from a finite domain) for the two predicates.

An interpretation is called a **model** for a *specific set of rules* if those rules are *always true* under that interpretation; that is, for any values assigned to the variables in the rules, the head of the rules is true when we substitute the truth values assigned to

- 
- |   |                           |
|---|---------------------------|
| 1. SUPERIOR(X, Y) :- SUPERVISE(X, Y).                 | (rule 1)                  |
| 2. SUPERIOR(X, Y) :- SUPERVISE(X, Z), SUPERIOR(Z, Y). | (rule 2)                  |
| 3. SUPERVISE(jennifer, ahmad).                        | (ground axiom, given)     |
| 4. SUPERVISE(james, jennifer).                        | (ground axiom, given)     |
| 5. SUPERIOR(jennifer, ahmad).                         | (apply rule 1 on 3)       |
| 6. SUPERIOR(james, ahmad).                            | (apply rule 2 on 4 and 5) |

**Figure 26.12**  
Proving a new fact.

---

<sup>33</sup>The most commonly chosen domain is finite and is called the *Herbrand Universe*.



the predicates in the body of the rule by that interpretation. Hence, whenever a particular substitution (binding) to the variables in the rules is applied, if all the predicates in the body of a rule are true under the interpretation, the predicate in the head of the rule must also be true. The interpretation shown in Figure 26.13 is a model for the two rules shown, since it can never cause the rules to be violated. Notice that a rule is violated if a particular binding of constants to the variables makes all the predicates in the rule body true but makes the predicate in the rule head false. For example, if  $\text{SUPERVISE}(a, b)$  and  $\text{SUPERIOR}(b, c)$  are both true under some interpretation, but  $\text{SUPERIOR}(a, c)$  is not true, the interpretation cannot be a model for the recursive rule:

$$\text{SUPERIOR}(X, Y) \text{ :- SUPERVISE}(X, Z), \text{SUPERIOR}(Z, Y)$$

In the model-theoretic approach, the meaning of the rules is established by providing a model for these rules. A model is called a **minimal model** for a set of rules if we cannot change any fact from true to false and still get a model for these rules. For

**Figure 26.13**

An interpretation that is a minimal model.

#### Rules

$\text{SUPERIOR}(X, Y) \text{ :- SUPERVISE}(X, Y).$

$\text{SUPERIOR}(X, Y) \text{ :- SUPERVISE}(X, Z), \text{SUPERIOR}(Z, Y).$

#### Interpretation

##### *Known Facts:*

$\text{SUPERVISE}(\text{franklin}, \text{john})$  is **true**.

$\text{SUPERVISE}(\text{franklin}, \text{ramesh})$  is **true**.

$\text{SUPERVISE}(\text{franklin}, \text{joyce})$  is **true**.

$\text{SUPERVISE}(\text{jennifer}, \text{alicia})$  is **true**.

$\text{SUPERVISE}(\text{jennifer}, \text{ahmad})$  is **true**.

$\text{SUPERVISE}(\text{james}, \text{franklin})$  is **true**.

$\text{SUPERVISE}(\text{james}, \text{jennifer})$  is **true**.

$\text{SUPERVISE}(X, Y)$  is **false** for all other possible  $(X, Y)$  combinations

##### *Derived Facts:*

$\text{SUPERIOR}(\text{franklin}, \text{john})$  is **true**.

$\text{SUPERIOR}(\text{franklin}, \text{ramesh})$  is **true**.

$\text{SUPERIOR}(\text{franklin}, \text{joyce})$  is **true**.

$\text{SUPERIOR}(\text{jennifer}, \text{alicia})$  is **true**.

$\text{SUPERIOR}(\text{jennifer}, \text{ahmad})$  is **true**.

$\text{SUPERIOR}(\text{james}, \text{franklin})$  is **true**.

$\text{SUPERIOR}(\text{james}, \text{jennifer})$  is **true**.

$\text{SUPERIOR}(\text{james}, \text{john})$  is **true**.

$\text{SUPERIOR}(\text{james}, \text{ramesh})$  is **true**.

$\text{SUPERIOR}(\text{james}, \text{joyce})$  is **true**.

$\text{SUPERIOR}(\text{james}, \text{alicia})$  is **true**.

$\text{SUPERIOR}(\text{james}, \text{ahmad})$  is **true**.

$\text{SUPERIOR}(X, Y)$  is **false** for all other possible  $(X, Y)$  combinations

example, consider the interpretation in Figure 26.13, and assume that the SUPERVISE predicate is defined by a set of known facts, whereas the SUPERIOR predicate is defined as an interpretation (model) for the rules. Suppose that we add the predicate SUPERIOR(james, bob) to the true predicates. This remains a model for the rules shown, but it is not a minimal model, since changing the truth value of SUPERIOR(james, bob) from true to false still provides us with a model for the rules. The model shown in Figure 26.13 is the minimal model for the set of facts that are defined by the SUPERVISE predicate.

In general, the minimal model that corresponds to a given set of facts in the model-theoretic interpretation should be the same as the facts generated by the proof-theoretic interpretation for the same original set of ground and deductive axioms. However, this is generally true only for rules with a simple structure. Once we allow negation in the specification of rules, the correspondence between interpretations *does not* hold. In fact, with negation, numerous minimal models are possible for a given set of facts.

A third approach to interpreting the meaning of rules involves defining an inference mechanism that is used by the system to deduce facts from the rules. This inference mechanism would define a **computational interpretation** to the meaning of the rules. The Prolog logic programming language uses its inference mechanism to define the meaning of the rules and facts in a Prolog program. Not all Prolog programs correspond to the proof-theoretic or model-theoretic interpretations; it depends on the type of rules in the program. However, for many simple Prolog programs, the Prolog inference mechanism infers the facts that correspond either to the proof-theoretic interpretation or to a minimal model under the model-theoretic interpretation.

### 26.5.6 Datalog Programs and Their Safety

There are two main methods of defining the truth values of predicates in actual Datalog programs. **Fact-defined predicates** (or **relations**) are defined by listing all the combinations of values (the tuples) that make the predicate true. These correspond to base relations whose contents are stored in a database system. Figure 26.14 shows the fact-defined predicates EMPLOYEE, MALE, FEMALE, DEPARTMENT, SUPERVISE, PROJECT, and WORKS\_ON, which correspond to part of the relational database shown in Figure 5.6. **Rule-defined predicates** (or **views**) are defined by being the head (LHS) of one or more Datalog rules; they correspond to *virtual relations* whose contents can be inferred by the inference engine. Figure 26.15 shows a number of rule-defined predicates.

A program or a rule is said to be **safe** if it generates a *finite* set of facts. The general theoretical problem of determining whether a set of rules is safe is undecidable. However, one can determine the safety of restricted forms of rules. For example, the rules shown in Figure 26.16 are safe. One situation where we get unsafe rules that can generate an infinite number of facts arises when one of the variables in the rule can range over an infinite domain of values, and that variable is not limited to ranging over a finite relation. For example, consider the following rule:

```
BIG_SALARY(Y) :- Y > 60000
```

**Figure 26.14**

Fact-defined  
predicates for part  
of the database from  
Figure 5.6.

EMPLOYEE(john).	MALE(john).
EMPLOYEE(franklin).	MALE(franklin).
EMPLOYEE(alicia).	MALE(ramesh).
EMPLOYEE(jennifer).	MALE(ahmad).
EMPLOYEE(ramesh).	MALE(james).
EMPLOYEE(joyce).	
EMPLOYEE(ahmad).	FEMALE(alicia).
EMPLOYEE(james).	FEMALE(jennifer).
	FEMALE(joyce).
SALARY(john, 30000).	
SALARY(franklin, 40000).	PROJECT(productx).
SALARY(alicia, 25000).	PROJECT(producty).
SALARY(jennifer, 43000).	PROJECT(productz).
SALARY(ramesh, 38000).	PROJECT(computerization).
SALARY(joyce, 25000).	PROJECT(reorganization).
SALARY(ahmad, 25000).	PROJECT(newbenefits).
SALARY(james, 55000).	
DEPARTMENT(john, research).	WORKS_ON(john, productx, 32).
DEPARTMENT(franklin, research).	WORKS_ON(john, producty, 8).
DEPARTMENT(alicia, administration).	WORKS_ON(ramesh, productz, 40).
DEPARTMENT(jennifer, administration).	WORKS_ON(joyce, productx, 20).
DEPARTMENT(ramesh, research).	WORKS_ON(joyce, producty, 20).
DEPARTMENT(joyce, research).	WORKS_ON(franklin, producty, 10).
DEPARTMENT(ahmad, administration).	WORKS_ON(franklin, productz, 10).
DEPARTMENT(james, headquarters).	WORKS_ON(franklin, computerization, 10).
	WORKS_ON(franklin, reorganization, 10).
SUPERVISE(franklin, john).	WORKS_ON(alicia, newbenefits, 30).
SUPERVISE(franklin, ramesh).	WORKS_ON(alicia, computerization, 10).
SUPERVISE(franklin, joyce).	WORKS_ON(ahmad, computerization, 35).
SUPERVISE(jennifer, alicia).	WORKS_ON(ahmad, newbenefits, 5).
SUPERVISE(jennifer, ahmad).	WORKS_ON(jennifer, newbenefits, 20).
SUPERVISE(james, franklin).	WORKS_ON(jennifer, reorganization, 15).
SUPERVISE(james, jennifer).	WORKS_ON(james, reorganization, 10).

**Figure 26.15**

Rule-defined  
predicates.

SUPERIOR( $X, Y$ ) :- SUPERVISE( $X, Y$ ).  
 SUPERIOR( $X, Y$ ) :- SUPERVISE( $X, Z$ ), SUPERIOR( $Z, Y$ ).  
 SUBORDINATE( $X, Y$ ) :- SUPERIOR( $Y, X$ ).  
 SUPERVISOR( $X$ ) :- EMPLOYEE( $X$ ), SUPERVISE( $X, Y$ ).  
 OVER\_40K\_EMP( $X$ ) :- EMPLOYEE( $X$ ), SALARY( $X, Y$ ),  $Y \geq 40000$ .  
 UNDER\_40K\_SUPERVISOR( $X$ ) :- SUPERVISOR( $X$ ), NOT(OVER\_40K\_EMP( $X$ )).  
 MAIN\_PRODUCTX\_EMP( $X$ ) :- EMPLOYEE( $X$ ), WORKS\_ON( $X$ , productx,  $Y$ ),  $Y \geq 20$ .  
 PRESIDENT( $X$ ) :- EMPLOYEE( $X$ ), NOT(SUPERVISE( $Y, X$ )).

```

REL_ONE(A, B, C).
REL_TWO(D, E, F).
REL_THREE(G, H, I, J).

SELECT_ONE_A_EQ_C(X, Y, Z) :- REL_ONE(C, Y, Z).
SELECT_ONE_B_LESS_5(X, Y, Z) :- REL_ONE(X, Y, Z), Y < 5.
SELECT_ONE_A_EQ_C_AND_B_LESS_5(X, Y, Z) :- REL_ONE(C, Y, Z), Y < 5.

SELECT_ONE_A_EQ_C_OR_B_LESS_5(X, Y, Z) :- REL_ONE(C, Y, Z).
SELECT_ONE_A_EQ_C_OR_B_LESS_5(X, Y, Z) :- REL_ONE(X, Y, Z), Y < 5.

PROJECT_THREE_ON_G_H(W, X) :- REL_THREE(W, X, Y, Z).

UNION_ONE_TWO(X, Y, Z) :- REL_ONE(X, Y, Z).
UNION_ONE_TWO(X, Y, Z) :- REL_TWO(X, Y, Z).

INTERSECT_ONE_TWO(X, Y, Z) :- REL_ONE(X, Y, Z), REL_TWO(X, Y, Z).

DIFFERENCE_TWO_ONE(X, Y, Z) :- _TWO(X, Y, Z) NOT(REL_ONE(X, Y, Z)).

CART_PROD_ONE_THREE(T, U, V, W, X, Y, Z) :-
    REL_ONE(T, U, V), REL_THREE(W, X, Y, Z).

NATURAL_JOIN_ONE_THREE_C_EQ_G(U, V, W, X, Y, Z) :-
    REL_ONE(U, V, W), REL_THREE(W, X, Y, Z).

```

**Figure 26.16**

Predicates for illustrating relational operations.

Here, we can get an infinite result if  $Y$  ranges over all possible integers. But suppose that we change the rule as follows:

```
BIG_SALARY(Y) :- EMPLOYEE(X), Salary(X, Y), Y > 60000
```

In the second rule, the result is not infinite, since the values that  $Y$  can be bound to are now restricted to values that are the salary of some employee in the database—presumably, a finite set of values. We can also rewrite the rule as follows:

```
BIG_SALARY(Y) :- Y > 60000, EMPLOYEE(X), Salary(X, Y)
```

In this case, the rule is still theoretically safe. However, in Prolog or any other system that uses a top-down, depth-first inference mechanism, the rule creates an infinite loop, since we first search for a value for  $Y$  and then check whether it is a salary of an employee. The result is generation of an infinite number of  $Y$  values, even though these, after a certain point, cannot lead to a set of true RHS predicates. One definition of Datalog considers both rules to be safe, since it does not depend on a particular inference mechanism. Nonetheless, it is generally advisable to write such a rule in the safest form, with the predicates that restrict possible bindings of variables placed first. As another example of an unsafe rule, consider the following rule:

```
HAS_SOMETHING(X, Y) :- EMPLOYEE(X)
```

Here, an infinite number of  $Y$  values can again be generated, since the variable  $Y$  appears only in the head of the rule and hence is not limited to a finite set of values. To define safe rules more formally, we use the concept of a limited variable. A variable  $X$  is **limited** in a rule if (1) it appears in a regular (not built-in) predicate in the body of the rule; (2) it appears in a predicate of the form  $X = c$  or  $c = X$  or  $(c_1 \leq X$  and  $X \leq c_2)$  in the rule body, where  $c$ ,  $c_1$ , and  $c_2$  are constant values; or (3) it appears in a predicate of the form  $X = Y$  or  $Y = X$  in the rule body, where  $Y$  is a limited variable. A rule is said to be **safe** if all its variables are limited.

### 26.5.7 Use of Relational Operations

It is straightforward to specify many operations of the relational algebra in the form of Datalog rules that define the result of applying these operations on the database relations (fact predicates). This means that relational queries and views can easily be specified in Datalog. The additional power that Datalog provides is in the specification of recursive queries, and views based on recursive queries. In this section, we show how some of the standard relational operations can be specified as Datalog rules. Our examples will use the base relations (fact-defined predicates) `REL_ONE`, `REL_TWO`, and `REL_THREE`, whose schemas are shown in Figure 26.16. In Datalog, we do not need to specify the attribute names as in Figure 26.16; rather, the arity (degree) of each predicate is the important aspect. In a practical system, the domain (data type) of each attribute is also important for operations such as `UNION`, `INTERSECTION`, and `JOIN`, and we assume that the attribute types are compatible for the various operations, as discussed in Chapter 3.

Figure 26.16 illustrates a number of basic relational operations. Notice that if the Datalog model is based on the relational model and hence assumes that predicates (fact relations and query results) specify sets of tuples, duplicate tuples in the same predicate are automatically eliminated. This may or may not be true, depending on the Datalog inference engine. However, it is definitely *not* the case in Prolog, so any of the rules in Figure 26.16 that involve duplicate elimination are not correct for Prolog. For example, if we want to specify Prolog rules for the `UNION` operation with duplicate elimination, we must rewrite them as follows:

```
UNION_ONE_TWO(X, Y, Z) :- REL_ONE(X, Y, Z).
UNION_ONE_TWO(X, Y, Z) :- REL_TWO(X, Y, Z), NOT(REL_ONE(X, Y, Z)).
```

However, the rules shown in Figure 26.16 should work for Datalog, if duplicates are automatically eliminated. Similarly, the rules for the `PROJECT` operation shown in Figure 26.16 should work for Datalog in this case, but they are not correct for Prolog, since duplicates would appear in the latter case.

### 26.5.8 Evaluation of Nonrecursive Datalog Queries

In order to use Datalog as a deductive database system, it is appropriate to define an inference mechanism based on relational database query processing concepts. The inherent strategy involves a bottom-up evaluation, starting with base relations; the order of operations is kept flexible and subject to query optimization. In this section we discuss an **inference mechanism** based on relational operations that can be

applied to **nonrecursive** Datalog queries. We use the fact and rule base shown in Figures 26.14 and 26.15 to illustrate our discussion.

If a query involves only fact-defined predicates, the inference becomes one of searching among the facts for the query result. For example, a query such as

DEPARTMENT( $X$ , Research)?

is a selection of all employee names  $X$  who work for the Research department. In relational algebra, it is the query:

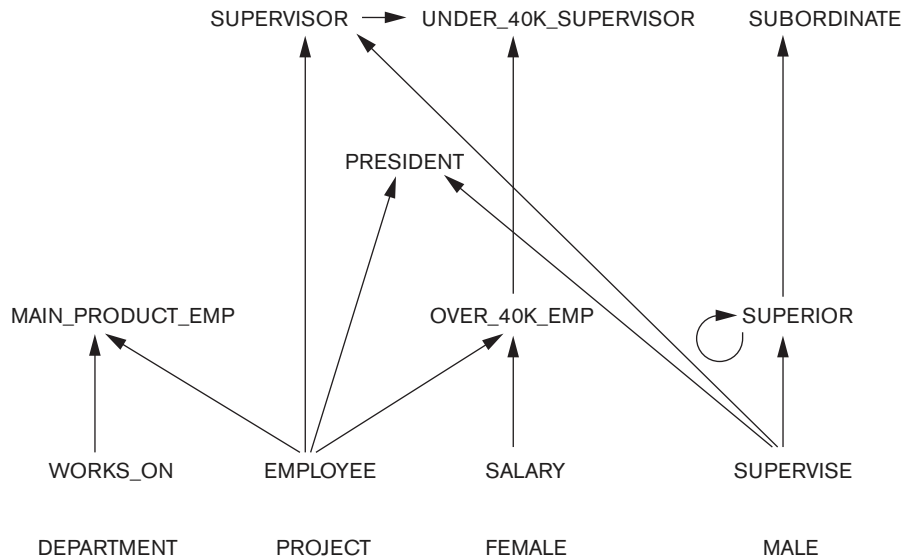
$\pi_{\$1} (\sigma_{\$2 = \text{"Research"}} (\text{DEPARTMENT}))$

which can be answered by searching through the fact-defined predicate department( $X$ ,  $Y$ ). The query involves relational SELECT and PROJECT operations on a base relation, and it can be handled by the database query processing and optimization techniques discussed in Chapter 19.

When a query involves rule-defined predicates, the inference mechanism must compute the result based on the rule definitions. If a query is nonrecursive and involves a predicate  $p$  that appears as the head of a rule  $p :- p_1, p_2, \dots, p_n$ , the strategy is first to compute the relations corresponding to  $p_1, p_2, \dots, p_n$  and then to compute the relation corresponding to  $p$ . It is useful to keep track of the dependency among the predicates of a deductive database in a **predicate dependency graph**. Figure 26.17 shows the graph for the fact and rule predicates shown in Figures 26.14 and 26.15. The dependency graph contains a **node** for each predicate. Whenever a predicate  $A$  is specified in the body (RHS) of a rule, and the head (LHS) of that rule is the predicate  $B$ , we say that  $B$  **depends on**  $A$ , and we draw a directed edge from  $A$  to  $B$ . This indicates that in order to compute the facts for the predicate  $B$  (the rule head), we must first compute the facts for all the predicates  $A$  in the rule body. If the dependency graph has no cycles, we call the rule set **nonrecursive**. If there is at least one cycle, we call the rule set **recursive**. In Figure 26.17, there is one recursively defined predicate—namely, SUPERIOR—which has a recursive edge pointing back to itself. Additionally, because the predicate subordinate depends on SUPERIOR, it also requires recursion in computing its result.

A query that includes only nonrecursive predicates is called a **nonrecursive query**. In this section we discuss only inference mechanisms for nonrecursive queries. In Figure 26.17, any query that does not involve the predicates SUBORDINATE or SUPERIOR is nonrecursive. In the predicate dependency graph, the nodes corresponding to fact-defined predicates do not have any incoming edges, since all fact-defined predicates have their facts stored in a database relation. The contents of a fact-defined predicate can be computed by directly retrieving the tuples in the corresponding database relation.

The main function of an inference mechanism is to compute the facts that correspond to query predicates. This can be accomplished by generating a **relational expression** involving relational operators as SELECT, PROJECT, JOIN, UNION, and SET DIFFERENCE (with appropriate provision for dealing with safety issues) that, when executed, provides the query result. The query can then be executed by utilizing the internal query processing and optimization operations of a relational database



**Figure 26.17**  
 Predicate dependency  
 graph for Figures 26.15  
 and 26.16.

management system. Whenever the inference mechanism needs to compute the fact set corresponding to a nonrecursive rule-defined predicate  $p$ , it first locates all the rules that have  $p$  as their head. The idea is to compute the fact set for each such rule and then to apply the UNION operation to the results, since UNION corresponds to a logical OR operation. The dependency graph indicates all predicates  $q$  on which each  $p$  depends, and since we assume that the predicate is nonrecursive, we can always determine a partial order among such predicates  $q$ . Before computing the fact set for  $p$ , first we compute the fact sets for all predicates  $q$  on which  $p$  depends, based on their partial order. For example, if a query involves the predicate UNDER\_40K\_SUPERVISOR, we must first compute both SUPERVISOR and OVER\_40K\_EMP. Since the latter two depend only on the fact-defined predicates EMPLOYEE, SALARY, and SUPERVISE, they can be computed directly from the stored database relations.

This concludes our introduction to deductive databases. Additional material may be found at the book's Web site, where the complete Chapter 25 from the third edition is available. Information on the Web site includes a discussion on algorithms for recursive query processing. We have included an extensive bibliography of work in deductive databases, recursive query processing, magic sets, combination of relational databases with deductive rules, and GLUE-NAIL! System, at the end of this chapter.

## 26.6 Summary

In this chapter, we introduced database concepts for some of the common features that are needed by advanced applications: active databases, temporal databases, spatial databases, multimedia databases, and deductive databases. It is important to note that each of these is a broad topic and warrants a complete textbook.



First in Section 26.1 we introduced the topic of active databases, which provide additional functionality for specifying active rules. We introduced the event-condition-action (ECA) model for active databases. The rules can be automatically triggered by events that occur—such as a database update—and they can initiate certain actions that have been specified in the rule declaration if certain conditions are true. Many commercial packages have some of the functionality provided by active databases in the form of triggers. We gave examples of row-level triggers in the Oracle commercial system in Section 26.1.1. We discussed the different options for specifying triggers in Section 26.1.2, such as row-level versus statement-level, before versus after, and immediate versus deferred. Then in Section 26.1.3 we gave examples of statement-level rules in the STARBURST experimental system. We briefly discussed some design issues and some possible applications for active databases in Section 26.1.4. The syntax for triggers in the SQL-99 standard was also discussed in Section 26.1.5.

Next in Section 26.2 we introduced some of the concepts of temporal databases, which permit the database system to store a history of changes and allow users to query both current and past states of the database. In Section 26.2.1, we discussed how time is represented and distinguished between the valid time and transaction time dimensions. In Section 26.2.2 we discussed how valid time, transaction time, and bitemporal relations can be implemented using tuple versioning in the relational model, and we provided examples to illustrate how updates, inserts, and deletes are implemented. We also showed how complex objects can be used to implement temporal databases using attribute versioning in Section 26.2.3. We looked at some of the querying operations for temporal relational databases and gave a brief introduction to the TSQL2 language in Section 26.2.4.

Then we turned to spatial databases in Section 26.3. Spatial databases provide concepts for databases that keep track of objects that have spatial characteristics. We gave an introduction to spatial databases in Section 26.3.1. We discussed the types of spatial data and spatial data models in Section 26.3.2, then the types of operators for processing spatial data and the types of spatial queries in Section 26.3.3. In Section 26.3.4, we gave an overview of spatial indexing techniques, including the popular *R*-trees. Then we introduced some spatial data mining techniques in Section 26.3.5, and discussed some applications that require spatial databases in Section 26.3.6.

In Section 26.4 we discussed some basic types of multimedia databases and their important characteristics. Multimedia databases provide features that allow users to store and query different types of multimedia information, which includes images (such as pictures and drawings), video clips (such as movies, newsreels, and home videos), audio clips (such as songs, phone messages, and speeches), and documents (such as books and articles). We provided a brief overview of the various types of media sources and how multimedia sources may be indexed. Images are an extremely common type of data among databases today and are likely to occupy a large proportion of stored data in databases. We therefore provided a more detailed treatment of images: their automatic analysis (Section 26.4.1), recognition of objects within images (Section 26.4.2), and their semantic tagging (Section 26.1.3)—all of which contribute to developing better systems to retrieve images by content, which



still remains a challenging problem. We also commented on the analysis of audio data sources in Section 26.4.4.

We concluded the chapter with an introduction to deductive databases in Section 26.5. We introduced deductive databases in Section 26.5.1, and gave an overview of Prolog and Datalog notation in Sections 26.5.2 and 26.5.3. We discussed the clausal form of formulas in Section 26.5.4. Datalog rules are restricted to Horn clauses, which contain at most one positive literal. We discussed the proof-theoretic and model-theoretic interpretation of rules in Section 26.5.5. We briefly discussed the safety of Datalog rules in Section 26.5.6 and the ways of expressing relational operators using Datalog rules in Section 26.5.7. Finally, we discussed an inference mechanism based on relational operations that can be used to evaluate nonrecursive Datalog queries using relational query optimization techniques in Section 26.5.8. Although Datalog has been a popular language with some applications, implementations of deductive database systems such as LDL or VALIDITY have not become widely commercially available.

## Review Questions

- 26.1.** What are the differences between row-level and statement-level active rules?
- 26.2.** What are the differences among immediate, deferred, and detached *consideration* of active rule conditions?
- 26.3.** What are the differences among immediate, deferred, and detached *execution* of active rule actions?
- 26.4.** Briefly discuss the consistency and termination problems when designing a set of active rules.
- 26.5.** Discuss some applications of active databases.
- 26.6.** Discuss how time is represented in temporal databases and compare the different time dimensions.
- 26.7.** What are the differences among valid time, transaction time, and bitemporal relations?
- 26.8.** Describe how the insert, delete, and update commands should be implemented on a valid time relation.
- 26.9.** Describe how the insert, delete, and update commands should be implemented on a bitemporal relation.
- 26.10.** Describe how the insert, delete, and update commands should be implemented on a transaction time relation.
- 26.11.** What are the main differences between tuple versioning and attribute versioning?
- 26.12.** How do spatial databases differ from regular databases?
- 26.13.** What are the different types of spatial data?

- 26.14. Name the main types of spatial operators and different classes of spatial queries.
- 26.15. What are the properties of *R*-trees that act as an index for spatial data?
- 26.16. Describe how a spatial join index between spatial objects can be constructed.
- 26.17. What are the different types of spatial data mining?
- 26.18. State the general form of a spatial association rule. Give an example of a spatial association rule.
- 26.19. What are the different types of multimedia sources?
- 26.20. How are multimedia sources indexed for content-based retrieval?
- 26.21. What important features of images are used to compare them?
- 26.22. What are the different approaches to recognizing objects in images?
- 26.23. How is semantic tagging of images used?
- 26.24. What are the difficulties in analyzing audio sources?
- 26.25. What are deductive databases?
- 26.26. Write sample rules in Prolog to define that courses with course number above CS5000 are graduate courses and that DBgrads are those graduate students who enroll in CS6400 and CS8803.
- 26.27. Define the clausal form of formulas and Horn clauses.
- 26.28. What is theorem proving, and what is proof-theoretic interpretation of rules?
- 26.29. What is model-theoretic interpretation and how does it differ from proof-theoretic interpretation?
- 26.30. What are fact-defined predicates and rule-defined predicates?
- 26.31. What is a safe rule?
- 26.32. Give examples of rules that can define relational operations SELECT, PROJECT, JOIN, and SET operations.
- 26.33. Discuss the inference mechanism based on relational operations that can be applied to evaluate nonrecursive Datalog queries.

## Exercises

- 26.34. Consider the COMPANY database described in Figure 5.6. Using the syntax of Oracle triggers, write active rules to do the following:
  - a. Whenever an employee's project assignments are changed, check if the total hours per week spent on the employee's projects are less than 30 or greater than 40; if so, notify the employee's direct supervisor.

**SALES**

<u>S_id</u>	<u>V_id</u>	Commission
-------------	-------------	------------

**Figure 26.18**

Database schema for sales and salesperson commissions in Exercise 26.36.

**SALES\_PERSON**

<u>Salesperson_id</u>	Name	Title	Phone	Sum_commissions
-----------------------	------	-------	-------	-----------------

- b. Whenever an employee is deleted, delete the PROJECT tuples and DEPENDENT tuples related to that employee, and if the employee manages a department or supervises employees, set the Mgr\_ssn for that department to NULL and set the Super\_ssn for those employees to NULL.
- 26.35.** Repeat Exercise 26.34 but use the syntax of STARBURST active rules.
- 26.36.** Consider the relational schema shown in Figure 26.18. Write active rules for keeping the Sum\_commissions attribute of SALES\_PERSON equal to the sum of the Commission attribute in SALES for each sales person. Your rules should also check if the Sum\_commissions exceeds 100000; if it does, call a procedure Notify\_manager(S\_id). Write both statement-level rules in STARBURST notation and row-level rules in Oracle.
- 26.37.** Consider the UNIVERSITY EER schema in Figure 4.10. Write some rules (in English) that could be implemented via active rules to enforce some common integrity constraints that you think are relevant to this application.
- 26.38.** Discuss which of the updates that created each of the tuples shown in Figure 26.9 were applied retroactively and which were applied proactively.
- 26.39.** Show how the following updates, if applied in sequence, would change the contents of the bitemporal EMP\_BT relation in Figure 26.9. For each update, state whether it is a retroactive or proactive update.
- a. On 2004-03-10,17:30:00, the salary of Narayan is updated to 40000, effective on 2004-03-01.
  - b. On 2003-07-30,08:31:00, the salary of Smith was corrected to show that it should have been entered as 31000 (instead of 30000 as shown), effective on 2003-06-01.
  - c. On 2004-03-18,08:31:00, the database was changed to indicate that Narayan was leaving the company (that is, logically deleted) effective on 2004-03-31.
  - d. On 2004-04-20,14:07:33, the database was changed to indicate the hiring of a new employee called Johnson, with the tuple <'Johnson', '334455667', 1, NULL > effective on 2004-04-20.
  - e. On 2004-04-28,12:54:02, the database was changed to indicate that Wong was leaving the company (that is, logically deleted) effective on 2004-06-01.
  - f. On 2004-05-05,13:07:33, the database was changed to indicate the rehiring of Brown, with the same department and supervisor but with salary 35000 effective on 2004-05-01.

**26.40.** Show how the updates given in Exercise 26.39, if applied in sequence, would change the contents of the valid time EMP\_VT relation in Figure 26.8.

**26.41.** Add the following facts to the sample database in Figure 26.11:

SUPERVISE(ahmad, bob), SUPERVISE(franklin, gwen)

First modify the supervisory tree in Figure 26.11(b) to reflect this change. Then construct a diagram showing the top-down evaluation of the query SUPERIOR(james, Y) using rules 1 and 2 from Figure 26.12.

**26.42.** Consider the following set of facts for the relation PARENT(X, Y), where Y is the parent of X:

PARENT(a, aa), PARENT(a, ab), PARENT(aa, aaa), PARENT(aa, aab),  
PARENT(aaa, aaaa), PARENT(aaa, aaab)

Consider the rules

$r_1$ : ANCESTOR(X, Y) :- PARENT(X, Y)

$r_2$ : ANCESTOR(X, Y) :- PARENT(X, Z), ANCESTOR(Z, Y)

which define ancestor Y of X as above.

a. Show how to solve the Datalog query

ANCESTOR(aa, X)?

and show your work at each step.

b. Show the same query by computing only the changes in the ancestor relation and using that in rule 2 each time.

[This question is derived from Bancilhon and Ramakrishnan (1986).]

**26.43.** Consider a deductive database with the following rules:

ANCESTOR(X, Y) :- FATHER(X, Y)

ANCESTOR(X, Y) :- FATHER(X, Z), ANCESTOR(Z, Y)

Notice that FATHER(X, Y) means that Y is the father of X; ANCESTOR(X, Y) means that Y is the ancestor of X.

Consider the following fact base:

FATHER(Harry, Issac), FATHER(Issac, John), FATHER(John, Kurt)

a. Construct a model-theoretic interpretation of the above rules using the given facts.

b. Consider that a database contains the above relations FATHER(X, Y), another relation BROTHER(X, Y), and a third relation BIRTH(X, B), where B is the birth date of person X. State a rule that computes the first cousins of the following variety: their fathers must be brothers.

c. Show a complete Datalog program with fact-based and rule-based literals that computes the following relation: list of pairs of cousins, where the first person is born after 1960 and the second after 1970. You may use *greater than* as a built-in predicate. (Note: Sample facts for brother, birth, and person must also be shown.)

**26.44.** Consider the following rules:

$\text{REACHABLE}(X, Y) :- \text{FLIGHT}(X, Y)$

$\text{REACHABLE}(X, Y) :- \text{FLIGHT}(X, Z), \text{REACHABLE}(Z, Y)$

where  $\text{REACHABLE}(X, Y)$  means that city  $Y$  can be reached from city  $X$ , and  $\text{FLIGHT}(X, Y)$  means that there is a flight to city  $Y$  from city  $X$ .

a. Construct fact predicates that describe the following:

Los Angeles, New York, Chicago, Atlanta, Frankfurt, Paris, Singapore, Sydney are cities.

The following flights exist: LA to NY, NY to Atlanta, Atlanta to Frankfurt, Frankfurt to Atlanta, Frankfurt to Singapore, and Singapore to Sydney. (Note: No flight in reverse direction can be automatically assumed.)

b. Is the given data cyclic? If so, in what sense?

c. Construct a model-theoretic interpretation (that is, an interpretation similar to the one shown in Figure 26.13) of the above facts and rules.

d. Consider the query

$\text{REACHABLE}(\text{Atlanta}, \text{Sydney})?$

How will this query be executed? List the series of steps it will go through.

e. Consider the following rule-defined predicates:

$\text{ROUND-TRIP-REACHABLE}(X, Y) :-$

$\text{REACHABLE}(X, Y), \text{REACHABLE}(Y, X)$

$\text{DURATION}(X, Y, Z)$

Draw a predicate dependency graph for the above predicates. (Note:  $\text{DURATION}(X, Y, Z)$  means that you can take a flight from  $X$  to  $Y$  in  $Z$  hours.)

f. Consider the following query: What cities are reachable in 12 hours from Atlanta? Show how to express it in Datalog. Assume built-in predicates like  $\text{greater-than}(X, Y)$ . Can this be converted into a relational algebra statement in a straightforward way? Why or why not?

g. Consider the predicate  $\text{population}(X, Y)$ , where  $Y$  is the population of city  $X$ . Consider the following query: List all possible bindings of the predicate pair  $(X, Y)$ , where  $Y$  is a city that can be reached in two flights from city  $X$ , which has over 1 million people. Show this query in Datalog. Draw a corresponding query tree in relational algebraic terms.

## Selected Bibliography

The book by Zaniolo et al. (1997) consists of several parts, each describing an advanced database concept such as active, temporal, and spatial/text/multimedia databases. Widom and Ceri (1996) and Ceri and Fraternali (1997) focus on active database concepts and systems. Snodgrass (1995) describes the TSQL2 language and data model. Khoshafian and Baker (1996), Faloutsos (1996), and Subrahmanian (1998) describe multimedia database concepts. Tansel et al. (1993) is a collection of chapters on temporal databases. The temporal extensions to SQL:2011 are discussed in Kulkarni and Michels (2012).

STARBURST rules are described in Widom and Finkelstein (1990). Early work on active databases includes the HiPAC project, discussed in Chakravarthy et al. (1989) and Chakravarthy (1990). A glossary for temporal databases is given in Jensen et al. (1994). Snodgrass (1987) focuses on TQuel, an early temporal query language.

Temporal normalization is defined in Navathe and Ahmed (1989). Paton (1999) and Paton and Diaz (1999) survey active databases. Chakravarthy et al. (1994) describe SENTINEL and object-based active systems. Lee et al. (1998) discuss time series management.

The book by Shekhar and Chawla (2003) consists of all aspects of spatial databases including spatial data models, spatial storage and indexing, and spatial data mining. Scholl et al. (2001) is another textbook on spatial data management. Albrecht (1996) describes in detail the various GIS analysis operations. Clementini and Di Felice (1993) give a detailed description of the spatial operators. Güting (1994) describes the spatial data structures and querying languages for spatial database systems. Guttman (1984) proposed R-trees for spatial data indexing. Manolopoulos et al. (2005) is a book on the theory and applications of R-trees. Papadias et al. (2003) discuss query processing using R-trees for spatial networks. Ester et al. (2001) provide a comprehensive discussion on the algorithms and applications of spatial data mining. Koperski and Han (1995) discuss association rule discovery from geographic databases. Brinkhoff et al. (1993) provide a comprehensive overview of the usage of R-trees for efficient processing of spatial joins. Rotem (1991) describes spatial join indexes comprehensively. Shekhar and Xiong (2008) is a compilation of various sources that discuss different aspects of spatial database management systems and GIS. The density-based clustering algorithms DBSCAN and DENCLUE are proposed by Ester et al. (1996) and Hinnenberg and Gabriel (2007), respectively.

Multimedia database modeling has a vast amount of literature—it is difficult to point to all important references here. IBM's QBIC (Query By Image Content) system described in Niblack et al. (1998) was one of the first comprehensive approaches for querying images based on content. It is now available as a part of IBM's DB2 database image extender. Zhao and Grosky (2002) discuss content-based image retrieval. Carneiro and Vasconcelos (2005) present a database-centric view of semantic image annotation and retrieval. Content-based retrieval of subimages is discussed by Luo and Nascimento (2004). Tuceryan and Jain (1998) discuss various aspects of texture analysis. Object recognition using SIFT is discussed in Lowe (2004). Lazebnik et al. (2004) describe the use of local affine regions to model 3D objects (RIFT). Among other object recognition approaches, G-RIF is described in Kim et al. (2006). Bay et al. (2006) discuss SURF, Ke and Sukthankar (2004) present PCA-SIFT, and Mikolajczyk and Schmid (2005) describe GLOH. Fan et al. (2004) present a technique for automatic image annotation by using concept-sensitive objects. Fotouhi et al. (2007) was the first international workshop on many faces of multimedia semantics, which is continuing annually. Thuraisingham (2001) classifies audio data into different categories and, by treating each of these categories differently, elaborates on the use of metadata for audio. Prabhakaran (1996) has also discussed how speech processing techniques can add valuable metadata information to the audio piece.

The early developments of the logic and database approach are surveyed by Gallaire et al. (1984). Reiter (1984) provides a reconstruction of relational database theory,

whereas Levesque (1984) provides a discussion of incomplete knowledge in light of logic. Gallaire and Minker (1978) provide an early book on this topic. A detailed treatment of logic and databases appears in Ullman (1989, Volume 2), and there is a related chapter in Volume 1 (1988). Ceri, Gottlob, and Tanca (1990) present a comprehensive yet concise treatment of logic and databases. Das (1992) is a comprehensive book on deductive databases and logic programming. The early history of Datalog is covered in Maier and Warren (1988). Clocksin and Mellish (2003) is an excellent reference on Prolog language.

Aho and Ullman (1979) provide an early algorithm for dealing with recursive queries, using the least fixed-point operator. Bancilhon and Ramakrishnan (1986) give an excellent and detailed description of the approaches to recursive query processing, with detailed examples of the naive and seminaive approaches. Excellent survey articles on deductive databases and recursive query processing include Warren (1992) and Ramakrishnan and Ullman (1995). A complete description of the seminaive approach based on relational algebra is given in Bancilhon (1985). Other approaches to recursive query processing include the recursive query/subquery strategy of Vieille (1986), which is a top-down interpreted strategy, and the Henschen-Naqvi (1984) top-down compiled iterative strategy. Balbin and Ramamohanrao (1987) discuss an extension of the seminaive differential approach for multiple predicates.

The original paper on magic sets is by Bancilhon et al. (1986). Beeri and Ramakrishnan (1987) extend it. Mumick et al. (1990a) show the applicability of magic sets to nonrecursive nested SQL queries. Other approaches to optimizing rules without rewriting them appear in Vieille (1986, 1987). Kifer and Lozinskii (1986) propose a different technique. Bry (1990) discusses how the top-down and bottom-up approaches can be reconciled. Whang and Navathe (1992) describe an extended disjunctive normal form technique to deal with recursion in relational algebra expressions for providing an expert system interface over a relational DBMS.

Chang (1981) describes an early system for combining deductive rules with relational databases. The LDL system prototype is described in Chimenti et al. (1990). Krishnamurthy and Naqvi (1989) introduce the *choice* notion in LDL. Zaniolo (1988) discusses the language issues for the LDL system. A language overview of CORAL is provided in Ramakrishnan et al. (1992), and the implementation is described in Ramakrishnan et al. (1993). An extension to support object-oriented features, called CORAL++, is described in Srivastava et al. (1993). Ullman (1985) provides the basis for the NAIL! system, which is described in Morris et al. (1987). Phipps et al. (1991) describe the GLUE-NAIL! deductive database system.

Zaniolo (1990) reviews the theoretical background and the practical importance of deductive databases. Nicolas (1997) gives an excellent history of the developments leading up to deductive object-oriented database (DOOD) systems. Falcone et al. (1997) survey the DOOD landscape. References on the VALIDITY system include Friesen et al. (1995), Vieille (1998), and Dietrich et al. (1999).



## Introduction to Information Retrieval and Web Search

In most of the chapters in this book so far, we have discussed techniques for modeling, designing, querying, transaction processing of, and managing *structured data*. In Section 13.1, we discussed the differences among structured, semistructured, and unstructured data. Information retrieval deals mainly with *unstructured data*, and the techniques for indexing, searching, and retrieving information from large collections of unstructured documents. In Chapter 24, on NOSQL technologies, we considered systems, like MongoDB, that are suited to handling data in the form of documents. In this chapter,<sup>1</sup> we will provide an introduction to information retrieval. This is a very broad topic, so we will focus on the similarities and differences between information retrieval and database technologies, and on the indexing techniques that form the basis of many information retrieval systems.

This chapter is organized as follows. In Section 27.1, we introduce information retrieval (IR) concepts and discuss how IR differs from traditional databases. Section 27.2 is devoted to a discussion of retrieval models, which form the basis for IR search. Section 27.3 covers different types of queries in IR systems. Section 27.4 discusses text preprocessing, and Section 27.5 provides an overview of IR indexing, which is at the heart of any IR system. In Section 27.6, we describe the various evaluation metrics for IR systems performance. Section 27.7 details Web analysis and its relationship to information retrieval, and Section 27.8 briefly introduces the current trends in IR. Section 27.9 summarizes the chapter. For a limited overview of IR, we suggest that students read Sections 27.1 through 27.6.

---

<sup>1</sup>This chapter is coauthored with Saurav Sahay, Intel Labs.



## 27.1 Information Retrieval (IR) Concepts

**Information retrieval** is the process of retrieving documents from a collection in response to a query (or a search request) by a user. This section provides an overview of IR concepts. In Section 27.1.1, we introduce information retrieval in general and then discuss the different kinds and levels of search that IR encompasses. In Section 27.1.2, we compare IR and database technologies. Section 27.1.3 gives a brief history of IR. We then present the different modes of user interaction with IR systems in Section 27.1.4. In Section 27.1.5, we describe the typical IR process with a detailed set of tasks and then with a simplified process flow, and we end with a brief discussion of digital libraries and the Web.

### 27.1.1 Introduction to Information Retrieval

We first review the distinction between structured and unstructured data (see Section 13.1) to see how information retrieval differs from structured data management. Consider a relation (or table) called HOUSES with the attributes:

HOUSES(Lot#, Address, Square\_footage, Listed\_price)

This is an example of *structured data*. We can compare this relation with home-buying contract documents, which are examples of *unstructured data*. These types of documents can vary from city to city, and even county to county, within a given state in the United States. Typically, a contract document in a particular state will have a standard list of clauses described in paragraphs within sections of the document, with some predetermined (fixed) text and some variable areas whose content is to be supplied by the specific buyer and seller. Other variable information would include interest rate for financing, down-payment amount, closing dates, and so on. The documents could also include pictures taken during a home inspection. The information content in such documents can be considered *unstructured data* that can be stored in a variety of possible arrangements and formats. By **unstructured information**, we generally mean information that does not have a well-defined formal model and corresponding formal language for representation and reasoning, but rather is based on understanding of natural language.

With the advent of the World Wide Web (or Web, for short), the volume of unstructured information stored in messages and documents that contain textual and multimedia information has exploded. These documents are stored in a variety of standard formats, including HTML, XML (see Chapter 13), and several audio and video formatting standards. Information retrieval deals with the problems of storing, indexing, and retrieving (searching) such information to satisfy the needs of users. The problems that IR deals with are exacerbated by the fact that the number of Web pages and the number of social interaction events is already in the billions and is growing at a phenomenal rate. All forms of unstructured data described above are being added at the rates of millions per day, expanding the searchable space on the Web at rapidly increasing rates.

Historically, **information retrieval** is “the discipline that deals with the structure, analysis, organization, storage, searching, and retrieval of information” as defined

by Gerald Salton, an IR pioneer.<sup>2</sup> We can enhance the definition slightly to say that it applies in the context of unstructured documents to satisfy a user's information needs. This field has existed even longer than the database field and was originally concerned with retrieval of cataloged information in libraries based on titles, authors, topics, and keywords. In academic programs, the field of IR has long been a part of Library and Information Science programs. Information in the context of IR does not require machine-understandable structures, such as in relational database systems. Examples of such information include written texts, abstracts, documents, books, Web pages, e-mails, instant messages, and collections from digital libraries. Therefore, all loosely represented (unstructured) or semistructured information is also part of the IR discipline.

We introduced XML modeling and retrieval in Chapter 13 and discussed advanced data types, including spatial, temporal, and multimedia data, in Chapter 26. RDBMS vendors are providing modules to support many of these data types, as well as XML data, in the newer versions of their products. These newer versions are sometimes referred to as *extended RDBMSs*, or *object-relational database management systems* (ORDBMSs; see Chapter 12). The challenge of dealing with unstructured data is largely an information retrieval problem, although database researchers have been applying database indexing and search techniques to some of these problems.

IR systems go beyond database systems in that they do not limit the user to a specific query language, nor do they expect the user to know the structure (schema) or content of a particular database. IR systems use a user's information need expressed as a **free-form search request** (sometimes called a **keyword search query**, or just **query**) for interpretation by the system. Whereas the IR field historically dealt with cataloging, processing, and accessing text in the form of documents for decades, in today's world the use of Web search engines is becoming the dominant way to find information. The traditional problems of text indexing and making collections of documents searchable have been transformed by making the Web itself into a quickly accessible repository of human knowledge or a virtual digital library.

An IR system can be characterized at different levels: by types of *users*, types of *data*, and the types of the *information need*, along with the size and scale of the information repository it addresses. Different IR systems are designed to address specific problems that require a combination of different characteristics. These characteristics can be briefly described as follows:

**Types of Users.** Users can greatly vary in their abilities to interact with computational systems. This ability depends on a multitude of factors, such as education, culture, and past exposure to computational environments. The user may be an *expert user* (for example, a curator or a librarian) who is searching for specific information that is clear in his/her mind, understands the scope and the structure of the available repository, and forms relevant queries for the task, or a *layperson user* with a generic information need. The latter cannot create highly relevant queries for search (for example, students trying to find information about a new topic, researchers trying to assimilate different points of

---

<sup>2</sup>See Salton's 1968 book entitled *Automatic Information Organization and Retrieval*.

view about a historical issue, a scientist verifying a claim by another scientist, or a person trying to shop for clothing). Designing systems suitable for different types of users is an important topic of IR that is typically studied in a field known as Human-Computer Information Retrieval.

**Types of Data.** Search systems can be tailored to specific types of data. For example, the problem of retrieving information about a specific topic may be handled more efficiently by customized search systems that are built to collect and retrieve only information related to that specific topic. The information repository could be hierarchically organized based on a concept or topic hierarchy. These topical *domain-specific* or *vertical IR systems* are not as large as or as diverse as the generic World Wide Web, which contains information on all kinds of topics. Given that these domain-specific collections exist and may have been acquired through a specific process, they can be exploited much more efficiently by a specialized system. Types of data can have different dimensions, such as *velocity*, *variety*, *volume*, and *veracity*. We discussed these in Section 25.1.

**Types of Information Need.** In the context of Web search, users' information needs may be defined as navigational, informational, or transactional.<sup>3</sup> **Navigational search** refers to finding a particular piece of information (such as the Georgia Tech University Web site) that a user needs quickly. The purpose of **informational search** is to find current information about a topic (such as research activities in the college of computing at Georgia Tech—this is the classic IR system task). The goal of **transactional search** is to reach a site where further interaction happens resulting in some transactional event (such as joining a social network, shopping for products, making online reservations, accessing databases, and so on).

**Levels of Scale.** In the words of Nobel Laureate Herbert Simon,

*“What information consumes is rather obvious: it consumes the attention of its recipients. Hence a wealth of information creates a poverty of attention, and a need to allocate that attention efficiently among the overabundance of information sources that might consume it.”*<sup>4</sup>

This overabundance of information sources in effect creates a high noise-to-signal ratio in IR systems. Especially on the Web, where billions of pages are indexed, IR interfaces are built with efficient scalable algorithms for distributed searching, indexing, caching, merging, and fault tolerance. IR search engines can be limited in level to more specific collections of documents. **Enterprise search systems** offer IR solutions for searching different entities in an enterprise's **intranet**, which consists of the network of computers within that enterprise. The searchable entities include e-mails, corporate documents, manuals, charts, and presentations, as well as reports related to people, meetings, and projects. Enterprise search systems still typically deal with hundreds of millions of entities in large global enterprises. On a smaller scale, there are personal information systems such as those on desktops and laptops, called

---

<sup>3</sup>See Broder (2002) for details.

<sup>4</sup>From Herbert A. Simon (1971), “Designing Organizations for an Information-Rich World.”

**desktop search engines** (for example, Google Desktop, OS X Spotlight), for retrieving files, folders, and different kinds of entities stored on the computer. There are other systems that use peer-to-peer technology, such as the BitTorrent protocol, which allows sharing of music in the form of audio files, as well as specialized search engines for audio, such as Lycos and Yahoo! audio search.

### 27.1.2 Databases and IR Systems: A Comparison

Within the computer science discipline, databases and IR systems are closely related fields. Databases deal with structured information retrieval through well-defined formal languages for representation and manipulation based on the theoretically founded data models. Efficient algorithms have been developed for operators that allow rapid execution of complex queries. IR, on the other hand, deals with unstructured search with possibly vague query or search semantics and without a well-defined logical schematic representation. Some of the key differences between databases and IR systems are listed in Table 27.1.

Whereas databases have fixed schemas defined in some data model such as the relational model, an IR system has no fixed data model; it views data or documents according to some scheme, such as the vector space model, to aid in query processing (see Section 27.2). Databases using the relational model employ SQL for queries and transactions. The queries are mapped into relational algebra operations and search algorithms (see Chapter 19) and return a new relation (table) as the query result, providing an exact answer to the query for the current state of the database. In IR systems, there is no fixed language for defining the structure (schema) of the document or for operating on the document—queries tend to be a set of query terms (keywords) or a free-form natural language phrase. An IR query result is a list of document id's, or some pieces of text or multimedia objects (images, videos, and so on), or a list of links to Web pages.

The result of a database query is an exact answer; if no matching records (tuples) are found in the relation, the result is empty (null). On the other hand, the answer

**Table 27.1** A Comparison of Databases and IR Systems

Databases	IR Systems
<ul style="list-style-type: none"> <li>■ Structured data</li> <li>■ Schema driven</li> <li>■ Relational (or object, hierarchical, and network) model is predominant</li> <li>■ Structured query model</li> <li>■ Rich metadata operations</li> <li>■ Query returns data</li> <li>■ Results are based on exact matching (always correct)</li> </ul>	<ul style="list-style-type: none"> <li>■ Unstructured data</li> <li>■ No fixed schema; various data models (e.g., vector space model)</li> <li>■ Free-form query models</li> <li>■ Rich data operations</li> <li>■ Search request returns list or pointers to documents</li> <li>■ Results are based on approximate matching and measures of effectiveness (may be imprecise and ranked)</li> </ul>

to a user request in an IR query represents the IR system's best attempt at retrieving the information most relevant to that query. Whereas database systems maintain a large amount of metadata and allow their use in query optimization, the operations in IR systems rely on the data values themselves and their occurrence frequencies. Complex statistical analysis is sometimes performed to determine the *relevance* of each document or parts of a document to the user request.

### 27.1.3 A Brief History of IR

Information retrieval has been a common task since the times of ancient civilizations, which devised ways to organize, store, and catalog documents and records. Media such as papyrus scrolls and stone tablets were used to record documented information in ancient times. These efforts allowed knowledge to be retained and transferred among generations. With the emergence of public libraries and the printing press, large-scale methods for producing, collecting, archiving, and distributing documents and books evolved. As computers and automatic storage systems emerged, the need to apply these methods to computerized systems arose. Several techniques emerged in the 1950s, such as the seminal work of H. P. Luhn,<sup>5</sup> who proposed using words and their frequency counts as indexing units for documents, and using measures of word overlap between queries and documents as the retrieval criterion. It was soon realized that storing large amounts of text was not difficult. The harder task was to search for and retrieve that information selectively for users with specific information needs. Methods that explored word distribution statistics gave rise to the choice of keywords based on their distribution properties<sup>6</sup> and also led to keyword-based weighting schemes.

The earlier experiments with document retrieval systems such as SMART<sup>7</sup> in the 1960s adopted the *inverted file organization* based on keywords and their weights as the method of indexing (see Section 17.6.4 on inverted indexing). Serial (or sequential) organization proved inadequate if queries required fast, near real-time response times. Proper organization of these files became an important area of study; document classification and clustering schemes ensued. The scale of retrieval experiments remained a challenge due to lack of availability of large text collections. This soon changed with the World Wide Web. Also, the Text Retrieval Conference (TREC) was launched by NIST (National Institute of Standards and Technology) in 1992 as a part of the TIPSTER program<sup>8</sup> with the goal of providing a platform for evaluating information retrieval methodologies and facilitating technology transfer to develop IR products.

A **search engine** is a practical application of information retrieval to large-scale document collections. With significant advances in computers and communications technologies, people today have interactive access to enormous amounts of user-generated

---

<sup>5</sup>See Luhn (1957) "A statistical approach to mechanized encoding and searching of literary information."

<sup>6</sup>See Salton, Yang, and Yu (1975).

<sup>7</sup>For details, see Buckley et al. (1993).

<sup>8</sup>For details, see Harman (1992).

distributed content on the Web. This has spurred the rapid growth in search engine technology, where search engines are trying to discover different kinds of real-time content found on the Web. The part of a search engine responsible for discovering, analyzing, and indexing these new documents is known as a **crawler**. Other types of search engines exist for specific domains of knowledge. For example, the biomedical literature search database was started in the 1970s and is now supported by the PubMed search engine,<sup>9</sup> which gives access to over 24 million abstracts.

Although continuous progress is being made to tailor search results to the needs of an end user, the challenge remains in providing high-quality, pertinent, and timely information that is precisely aligned to the information needs of individual users.

### 27.1.4 Modes of Interaction in IR Systems

In the beginning of Section 27.1, we defined *information retrieval* as the process of retrieving documents from a collection in response to a query (or a search request) by a user. Typically the collection is made up of documents containing unstructured data. Other kinds of documents include images, audio recordings, video strips, and maps. Data may be scattered nonuniformly in these documents with no definitive structure. A **query** is a set of **terms** (also referred to as **keywords**) used by the searcher to specify an information need (for example, the terms *databases* and *operating systems* may be regarded as a query to a computer science bibliographic database). An informational request or a search query may also be a natural language phrase or a question (for example, “What is the currency of China?” or “Find Italian restaurants in Sarasota, Florida.”).

There are two main modes of interaction with IR systems—retrieval and browsing—which, although similar in goal, are accomplished through different interaction tasks. **Retrieval** is concerned with the extraction of relevant information from a repository of documents through an IR query, whereas **browsing** signifies the exploratory activity of a user visiting or navigating through similar or related documents based on the user’s assessment of relevance. During browsing, a user’s information need may not be defined *a priori* and is flexible. Consider the following browsing scenario: A user specifies ‘Atlanta’ as a keyword. The information retrieval system retrieves links to relevant result documents containing various aspects of Atlanta for the user. The user comes across the term ‘Georgia Tech’ in one of the returned documents and uses some access technique (such as clicking on the phrase ‘Georgia Tech’ in a document that has a built-in link) and visits documents about Georgia Tech in the same or a different Web site (repository). There the user finds an entry for ‘Athletics’ that leads the user to information about various athletic programs at Georgia Tech. Eventually, the user ends his search at the Fall schedule for the Yellow Jackets football team, which he finds to be of great interest. This user activity is known as browsing. **Hyperlinks** are used to interconnect Web pages and are mainly used for browsing. **Anchor texts** are text phrases within documents used to label hyperlinks and are very relevant to browsing.

<sup>9</sup>See [www.ncbi.nlm.nih.gov/pubmed/](http://www.ncbi.nlm.nih.gov/pubmed/)



**Web search** combines both aspects—browsing and retrieval—and is one of the main applications of information retrieval today. Web pages are analogous to documents. Web search engines maintain an indexed repository of Web pages, usually using the technique of inverted indexing (see Section 27.5). They retrieve the most relevant Web pages for the user in response to the user’s search request with a possible ranking in descending order of relevance. The **rank of a Web page** in a retrieved set is the measure of its relevance to the query that generated the result set.

### 27.1.5 Generic IR Pipeline

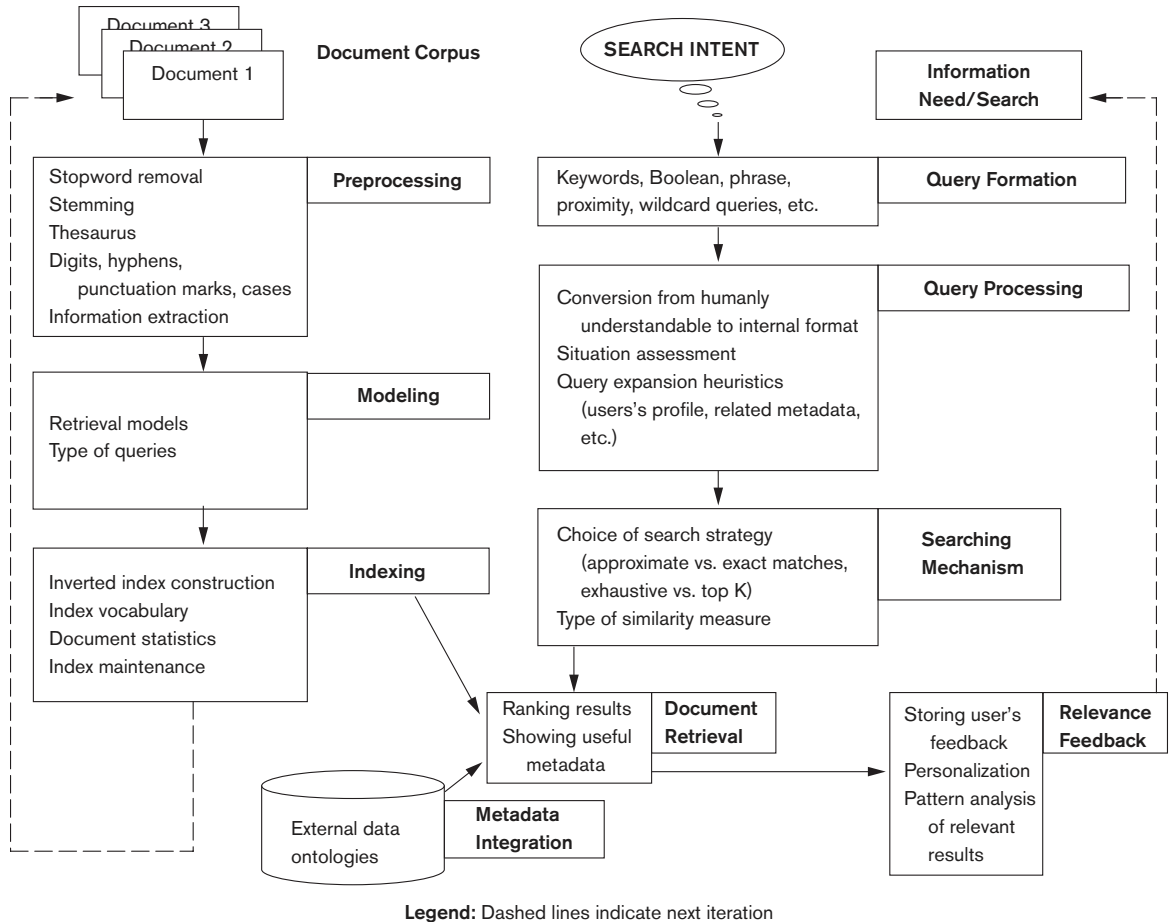
As we mentioned earlier, documents are made up of unstructured natural language text composed of character strings from English and other languages. Common examples of documents include newswire services (such as AP or Reuters), corporate manuals and reports, government notices, Web page articles, blogs, tweets, books, and journal papers. There are two main approaches to IR: statistical and semantic.

In a **statistical approach**, documents are analyzed and broken down into chunks of text (words, phrases, or  $n$ -grams, which are all subsequences of length  $n$  characters in a text or document) and each word or phrase is counted, weighted, and measured for relevance or importance. These words and their properties are then compared with the query terms for potential degree of match to produce a ranked list of resulting documents that contain the words. Statistical approaches are further classified based on the method employed. The three main statistical approaches are Boolean, vector space, and probabilistic (see Section 27.2).

**Semantic approaches** to IR use knowledge-based techniques of retrieval that broadly rely on the syntactic, lexical, sentential, discourse-based, and pragmatic levels of knowledge understanding. In practice, semantic approaches also apply some form of statistical analysis to improve the retrieval process.

Figure 27.1 shows the various stages involved in an IR processing system. The steps shown on the left in Figure 27.1 are typically offline processes, which prepare a set of documents for efficient retrieval; these are document preprocessing, document modeling, and indexing. The right side of Figure 27.1 deals with the process of a user interacting with the IR system either during a querying, browsing, or searching. It shows the steps involved; namely, query formation, query processing, searching mechanism, document retrieval, and relevance feedback. In each box, we highlight the important concepts and issues. The rest of this chapter describes some of the concepts involved in the various tasks within the IR process shown in Figure 27.1.

Figure 27.2 shows a simplified IR processing pipeline. In order to perform retrieval on documents, the documents are first represented in a form suitable for retrieval. The significant terms and their properties are extracted from the documents and are represented in a document index where the words/terms and their properties are stored in a matrix that contains each individual document in a row and each row contains the references to the words contained in those documents. This index is then converted into an inverted index (see Figure 27.4) of a word/term versus document matrix. Given the query words, the documents containing these words—



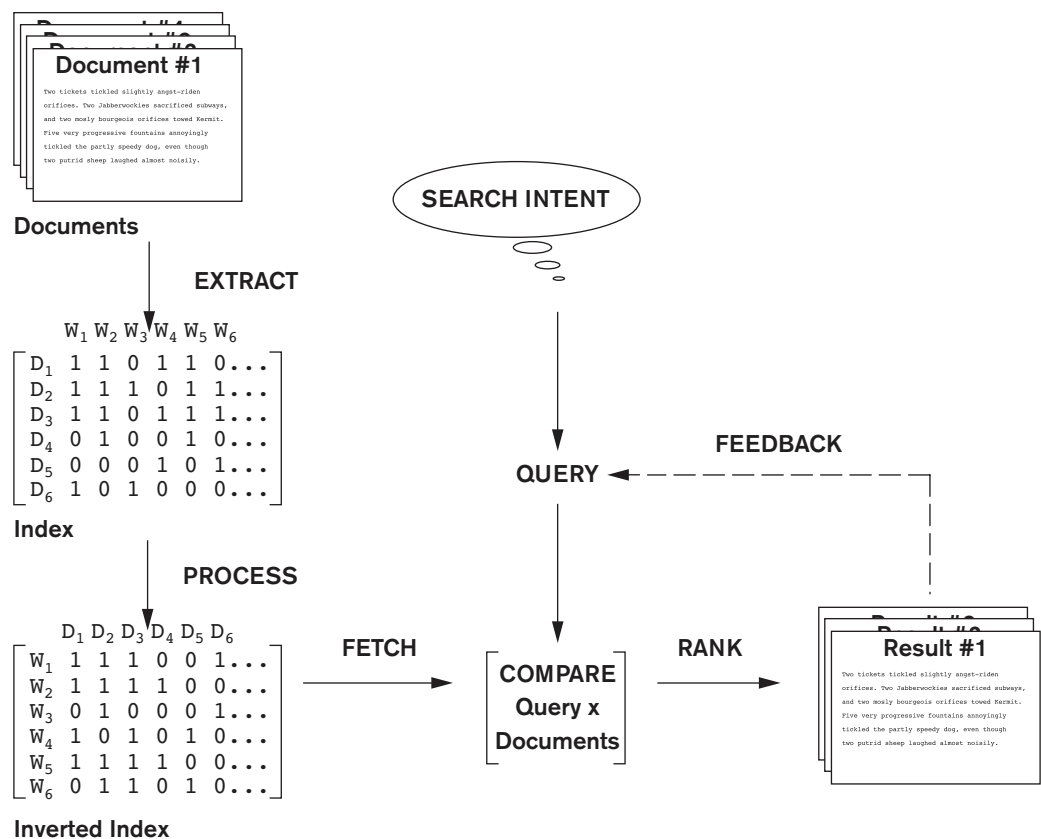
**Figure 27.1**  
Generic IR framework.

and the document properties, such as date of creation, author, and type of document—are fetched from the inverted index and compared with the query. This comparison results in a ranked list shown to the user. The user can then provide feedback on the results that triggers implicit or explicit query modification and expansion to fetch results that are more relevant for the user. Most IR systems allow for an interactive search in which the query and the results are successively refined.

## 27.2 Retrieval Models

In this section, we briefly describe the important models of IR. These are the three main statistical models—Boolean, vector space, and probabilistic—and the semantic model.





**Figure 27.2**  
Simplified IR process pipeline.

### 27.2.1 Boolean Model

In this model, documents are represented as a set of *terms*. Queries are formulated as a combination of terms using the standard Boolean logic set-theoretic operators such as AND, OR and NOT. Retrieval and relevance are considered as binary concepts in this model, so the retrieved elements are an “exact match” retrieval of relevant documents. There is no notion of ranking of resulting documents. All retrieved documents are considered equally important—a major simplification that does not consider frequencies of document terms or their proximity to other terms compared against the query terms.

Boolean retrieval models lack sophisticated ranking algorithms and are among the earliest and simplest information retrieval models. These models make it easy to associate metadata information and write queries that match the contents of the documents as well as other properties of documents, such as date of creation, author, and type of document.

### 27.2.2 Vector Space Model

The vector space model provides a framework in which term weighting, ranking of retrieved documents, and determining the relevance of feedback are possible. Using individual terms as dimensions, each document is represented by an  $n$ -dimensional vector of values. The values themselves may be a Boolean value to represent the existence or absence of the term in that document; alternately, they may be a number representative of the weight or frequency in the document. **Features** are a subset of the terms in a *set of documents* that are deemed most relevant to an IR search for this particular set of documents. The process of selecting these important terms (features) and their properties as a sparse (limited) list out of the very large number of available terms (the vocabulary can contain hundreds of thousands of terms) is independent of the model specification. The query is also specified as a terms vector (vector of features), and this is compared to the document vectors for similarity/relevance assessment.

The similarity assessment function that compares two vectors is not inherent to the model—different similarity functions can be used. However, the cosine of the angle between the query and document vector is a commonly used function for similarity assessment. As the angle between the vectors decreases, the cosine of the angle approaches one, meaning that the similarity of the query with a document vector increases. Terms (features) are weighted proportional to their frequency counts to reflect the importance of terms in the calculation of relevance measure. This is different from the Boolean model, which does not take into account the frequency of words in the document for relevance match.

In the vector model, the *document term weight*  $w_{ij}$  (for term  $i$  in document  $j$ ) is represented based on some variation of the TF (term frequency) or TF-IDF (term frequency–inverse document frequency) scheme (as we will describe below). **TF-IDF** is a statistical weight measure that is used to evaluate the importance of a document word in a collection of documents. The following formula is typically used:

$$\text{cosine}(d_j, q) = \frac{\langle d_j \times q \rangle}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{|V|} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{|V|} w_{ij}^2} \times \sqrt{\sum_{i=1}^{|V|} w_{iq}^2}}$$

In the formula given above, we use the following symbols:

- $d_j$  is the document vector for document  $j$ .
- $q$  is the query vector.
- $w_{ij}$  is the weight of term  $i$  in document  $j$ .
- $w_{iq}$  is the weight of term  $i$  in query vector  $q$ .
- $|V|$  is the number of dimensions in the vector that is the total number of important keywords (or features).

TF-IDF uses the product of normalized frequency of a term  $i$  ( $TF_{ij}$ ) in document  $D_j$  and the inverse document frequency of the term  $i$  ( $IDF_i$ ) to weight a term in a

document. The idea is that terms that capture the essence of a document occur frequently in the document (that is, their TF is high), but if such a term were to be a good term that discriminates the document from others, it must occur in only a few documents in the general population (that is, its IDF should be high as well).

IDF values can be easily computed for a fixed collection of documents. In case of Web search engines, taking a representative sample of documents approximates IDF computation. The following formulas can be used:

$$TF_{ij} = f_{ij} / \sum_{i=1 \text{ to } |V|} f_{ij}$$

$$IDF_i = \log(N / n_i)$$

In these formulas, the meaning of the symbols is:

- $TF_{ij}$  is the normalized term frequency of term  $i$  in document  $D_j$ .
- $f_{ij}$  is the number of occurrences of term  $i$  in document  $D_j$ .
- $IDF_i$  is the inverse document frequency weight for term  $i$ .
- $N$  is the number of documents in the collection.
- $n_i$  is the number of documents in which term  $i$  occurs.

Note that if a term  $i$  occurs in all documents, then  $n_i = N$  and hence  $IDF_i = \log(1)$  becomes zero, nullifying its importance and creating a situation where division by zero can occur. The weight of term  $i$  in document  $j$ ,  $w_{ij}$ , is computed based on its TF-IDF value in some techniques. To prevent division by zero, it is common to add a 1 to the denominator in the formulae such as the cosine formula above.

Sometimes, the relevance of the document with respect to a query ( $\text{rel}(D_j, Q)$ ) is directly measured as the sum of the TF-IDF values of the terms in the query  $Q$ :

$$\text{rel}(D_j, Q) = \sum_{i \in Q} TF_{ij} \times IDF_i$$

The normalization factor (similar to the denominator of the cosine formula) is incorporated into the TF-IDF formula itself, thereby measuring relevance of a document to the query by the computation of the dot product of the query and document vectors.

The Rocchio<sup>10</sup> algorithm is a well-known relevance feedback algorithm based on the vector space model that modifies the initial query vector and its weights in response to user-identified relevant documents. It expands the original query vector  $q$  to a new vector  $q_e$  as follows:

$$q_e = \alpha q + \frac{\beta}{|D_r|} \sum_{d_r \in D_r} d_r - \frac{\gamma}{|D_{nr}|} \sum_{d_{nr} \in D_{nr}} d_{nr},$$

---

<sup>10</sup>See Rocchio (1971).

Here,  $D_r$  stands for document-relevant ( $D_r$ ) and  $D_{nr}$  stands for document-nonrelevant ( $D_{nr}$ ); these terms represent relevant and nonrelevant document sets, respectively. Terms from relevant and nonrelevant documents get added to the original query vector with positive and negative weights, respectively, to create the modified query vector.  $\alpha$ ,  $\beta$ , and  $\gamma$  are parameters of the equation. The summation over  $d_r$  represents summation over all relevant terms of document  $d_r$ . Similarly, summation over  $d_{nr}$  represents summation over all nonrelevant terms of document  $d_{nr}$ . The values of these parameters determine how the feedback affects the original query, and these may be determined after a number of trial-and-error experiments.

### 27.2.3 Probabilistic Model

The similarity measures in the vector space model are somewhat ad hoc. For example, the model assumes that those documents closer to the query in cosine space are more relevant to the query vector. In the probabilistic model, a more concrete and definitive approach is taken: ranking documents by their estimated probability of relevance with respect to the query and the document. This is the basis of the *probability ranking principle* developed by Robertson.<sup>11</sup>

In the probabilistic framework, the IR system must decide whether the documents belong to the **relevant set** or the **nonrelevant set** for a query. To make this decision, it is assumed that a predefined relevant set and nonrelevant set exist for the query, and the task is to calculate the probability that the document belongs to the relevant set and compare that with the probability that the document belongs to the nonrelevant set.

Given the document representation  $D$  of a document, estimating the relevance  $R$  and nonrelevance  $NR$  of that document involves computation of conditional probability  $P(R|D)$  and  $P(NR|D)$ . These conditional probabilities can be calculated using Bayes' rule:<sup>12</sup>

$$\begin{aligned} P(R|D) &= P(D|R) \times P(R)/P(D) \\ P(NR|D) &= P(D|NR) \times P(NR)/P(D) \end{aligned}$$

A document  $D$  is classified as relevant if  $P(R|D) > P(NR|D)$ . Discarding the constant  $P(D)$ , this is equivalent to saying that a document is relevant if:

$$P(D|R) \times P(R) > P(D|NR) \times P(NR)$$

The likelihood ratio  $P(D|R)/P(D|NR)$  is used as a score to determine the likelihood of the document with representation  $D$  belonging to the relevant set.

The *term independence* or *naïve Bayes* assumption is used to estimate  $P(D|R)$  using computation of  $P(t_i|R)$  for term  $t_i$ . The likelihood ratios  $P(D|R)/P(D|NR)$  of documents are used as a proxy for ranking based on the assumption that highly ranked documents will have a high likelihood of belonging to the relevant set.<sup>13</sup>

<sup>11</sup>For a description of the Cheshire II system, see Robertson (1997).

<sup>12</sup>Bayes' theorem is a standard technique for measuring likelihood; see Howson and Urbach (1993), for example.

<sup>13</sup>Readers should refer to Croft et al. (2009) pages 246–247 for a detailed description.

With some reasonable assumptions and estimates about the probabilistic model along with extensions for incorporating query term weights and document term weights in the model, a probabilistic ranking algorithm called **BM25** (Best Match 25) is quite popular. This weighting scheme has evolved from several versions of the **Okapi**<sup>14</sup> system.

The Okapi weight for document  $d_j$  and query  $q$  is computed by the formula below. Additional notations are as follows:

- $t_i$  is a term.
- $f_{ij}$  is the raw frequency count of term  $t_i$  in document  $d_j$ .
- $f_{iq}$  is the raw frequency count of term  $t_i$  in query  $q$ .
- $N$  is the total number of documents in the collection.
- $df_i$  is the number of documents that contain the term  $t_i$ .
- $dl_j$  is the document length (in bytes) of  $d_j$ .
- $avdl$  is the average document length of the collection.

The Okapi relevance score of a document  $d_j$  for a query  $q$  is given by the equation below, where  $k_1$  (between 1.0–2.0),  $b$  (usually 0.75), and  $k_2$  (between 1–1,000) are parameters:

$$\text{okapi}(d_j, q) = \sum_{t_i \in q, d_j} \ln \frac{N - df_i + 0.5}{df_i + 0.5} \times \frac{(k_1 + 1)f_{ij}}{k_1 \left( 1 - b + b \frac{dl_j}{avdl} \right) + f_{ij}} \times \frac{(k_2 + 1)f_{iq}}{k_2 + f_{iq}}$$

### 27.2.4 Semantic Model

However sophisticated the above statistical models become, they can miss many relevant documents because those models do not capture the complete meaning or information need conveyed by a user's query. In semantic models, the process of matching documents to a given query is based on concept level and semantic matching instead of index term (keyword) matching. This allows retrieval of relevant documents that share meaningful associations with other documents in the query result, even when these associations are not inherently observed or statistically captured.

Semantic approaches include different levels of analysis, such as morphological, syntactic, and semantic analysis, to retrieve documents more effectively. In **morphological analysis**, roots and affixes are analyzed to determine the parts of speech (nouns, verbs, adjectives, and so on) of the words. Following morphological analysis, **syntactic analysis** follows to parse and analyze complete phrases in documents. Finally, the semantic methods have to resolve word ambiguities and/or generate relevant synonyms based on the **semantic relationships** among levels of structural entities in documents (words, paragraphs, pages, or entire documents).

<sup>14</sup>City University of London Okapi System by Robertson, Walker, and Hancock-Beaulieu (1995).

The development of a sophisticated semantic system requires complex knowledge bases of semantic information as well as retrieval heuristics. These systems often require techniques from artificial intelligence and expert systems. Knowledge bases like Cyc<sup>15</sup> and WordNet<sup>16</sup> have been developed for use in *knowledge-based IR systems* based on semantic models. The Cyc knowledge base, for example, is a representation of a vast quantity of commonsense knowledge. It presently contains 15.94 million assertions, 498,271 atomic concepts, and 441,159 nonatomic derived concepts for reasoning about the objects and events of everyday life. WordNet is an extensive thesaurus (over 117,000 concepts) that is very popular and is used by many systems and is under continuous development (see Section 27.4.3).

## 27.3 Types of Queries in IR Systems

Different keywords are associated with the document set during the process of indexing. These keywords generally consist of words, phrases, and other characterizations of documents such as date created, author names, and type of document. They are used by an IR system to build an inverted index (see Section 27.5), which is then consulted during the search. The queries formulated by users are compared to the set of index keywords. Most IR systems also allow the use of Boolean and other operators to build a complex query. The query language with these operators enriches the expressiveness of a user's information need.

### 27.3.1 Keyword Queries

Keyword-based queries are the simplest and most commonly used forms of IR queries: the user just enters keyword combinations to retrieve documents. The query keyword terms are implicitly connected by a logical AND operator. A query such as 'database concepts' retrieves documents that contain both the words 'database' and 'concepts' at the top of the retrieved results. In addition, most systems also retrieve documents that contain only 'database' or only 'concepts' in their text. Some systems remove most commonly occurring words (such as *a*, *the*, *of*, and so on, called **stopwords**) as a preprocessing step before sending the filtered query keywords to the IR engine. Most IR systems do not pay attention to the ordering of these words in the query. All retrieval models provide support for keyword queries.

### 27.3.2 Boolean Queries

Some IR systems allow using the AND, OR, NOT, ( ), +, and – Boolean operators in combinations of keyword formulations. AND requires that both terms be found. OR lets either term be found. NOT means any record containing the second term will be excluded. '( )' means the Boolean operators can be nested using parentheses. '+' is equivalent to AND, requiring the term; the '+' should be placed directly in front

---

<sup>15</sup>See Lenat (1995).

<sup>16</sup>See Miller (1990) for a detailed description of WordNet.

of the search term. ‘-’ is equivalent to AND NOT and means to exclude the term; the ‘-’ should be placed directly in front of the search term not wanted. Complex Boolean queries can be built out of these operators and their combinations, and they are evaluated according to the classical rules of Boolean algebra. No ranking is possible, because a document either satisfies such a query (is “relevant”) or does not satisfy it (is “nonrelevant”). A document is retrieved for a Boolean query if the query is logically true as an exact match in the document. Users generally do not use combinations of these complex Boolean operators, and IR systems support a restricted version of these set operators. Boolean retrieval models can directly support different Boolean operator implementations for these kinds of queries.

### 27.3.3 Phrase Queries

When documents are represented using an inverted keyword index for searching, the relative order of the terms in the document is lost. In order to perform exact phrase retrieval, these phrases should be encoded in the inverted index or implemented differently (with relative positions of word occurrences in documents). A phrase query consists of a sequence of words that makes up a phrase. The phrase is generally enclosed within double quotes. Each retrieved document must contain at least one instance of the exact phrase. Phrase searching is a more restricted and specific version of proximity searching that we mention below. For example, a phrase searching query could be ‘conceptual database design’. If phrases are indexed by the retrieval model, any retrieval model can be used for these query types. A phrase thesaurus may also be used in semantic models for fast dictionary searching of phrases.

### 27.3.4 Proximity Queries

Proximity search refers to a search that accounts for how close within a record multiple terms should be to each other. The most commonly used proximity search option is a phrase search that requires terms to be in the exact order. Other proximity operators can specify how close terms should be to each other. Some will also specify the order of the search terms. Each search engine can define proximity operators differently, and the search engines use various operator names such as NEAR, ADJ(adjacent), or AFTER. In some cases, a sequence of single words is given, together with a maximum allowed distance between them. Vector space models that also maintain information about positions and offsets of tokens (words) have robust implementations for this query type. However, providing support for complex proximity operators becomes computationally expensive because it requires the time-consuming preprocessing of documents and is thus suitable for smaller document collections rather than for the Web.

### 27.3.5 Wildcard Queries

Wildcard searching is generally meant to support regular expressions and pattern matching-based searching in text. In IR systems, certain kinds of wildcard search support may be implemented—usually words with any trailing characters (for example, ‘data\*’ would retrieve *data*, *database*, *datapoint*, *dataset*, and so on). Providing full support

for wildcard searches in Web search engines involves preprocessing overhead and is not generally implemented by many Web search engines today.<sup>17</sup> Retrieval models do not directly provide support for this query type. Lucene<sup>18</sup> provides support for certain types of wildcard queries. The query parser in Lucene computes a large Boolean query combining all combinations and expansions of words from the index.

### 27.3.6 Natural Language Queries

There are a few natural language search engines that aim to understand the structure and meaning of queries written in natural language text, generally as a question or narrative. This is an active area of research that employs techniques like shallow semantic parsing of text, or query reformulations based on natural language understanding. The system tries to formulate answers for such queries from retrieved results. Some search systems are starting to provide natural language interfaces to provide answers to specific types of questions, such as definition and factoid questions, which ask for definitions of technical terms or common facts that can be retrieved from specialized databases. Such questions are usually easier to answer because there are strong linguistic patterns giving clues to specific types of sentences—for example, ‘defined as’ or ‘refers to’. Semantic models can provide support for this query type.

## 27.4 Text Preprocessing

In this section, we review the commonly used text preprocessing techniques that are part of the text processing task in Figure 27.1.

### 27.4.1 Stopword Removal

**Stopwords** are very commonly used words in a language that play a major role in the formation of a sentence but that seldom contribute to the meaning of that sentence. Words that are expected to occur in 80% or more of the documents in a collection are typically referred to as *stopwords*, and they are rendered potentially useless. Because of the commonness and function of these words, they do not contribute much to the relevance of a document for a query search. Examples include words such as *the, of, to, a, and, in, said, for, that, was, on, he, is, with, at, by, and it*. These words are presented here with decreasing frequency of occurrence from a large corpus of documents called **AP89**.<sup>19</sup> The first six of these words account for 20% of all words in the listing, and the most frequent 50 words account for 40% of all text.

Removal of stopwords from a document must be performed before indexing. Articles, prepositions, conjunctions, and some pronouns are generally classified as stopwords. Queries must also be preprocessed for stopwords removal before the actual retrieval process. Removal of stopwords results in elimination of possible spurious

<sup>17</sup>See [http://www.livinginternet.com/w/wu\\_expert\\_wild.htm](http://www.livinginternet.com/w/wu_expert_wild.htm) for further details.

<sup>18</sup><http://lucene.apache.org/>

<sup>19</sup>For details, see Croft et al. (2009), pages 75–90.



indexes, thereby reducing the size of an index structure by about 40% or more. However, doing so could impact the recall if the stopword is an integral part of a query (for example, a search for the phrase ‘To be or not to be’, where removal of stopwords makes the query inappropriate, as all the words in the phrase are stopwords). Many search engines do not employ query stopword removal for this reason.

### 27.4.2 Stemming

A **stem** of a word is defined as the word obtained after trimming the suffix and prefix of an original word. For example, ‘comput’ is the stem word for *computer*, *computing*, *computable*, and *computation*. These suffixes and prefixes are very common in the English language for supporting the notion of verbs, tenses, and plural forms. **Stemming** reduces the different forms of the word formed by inflection (due to plurals or tenses) and derivation to a common stem.

A stemming algorithm can be applied to reduce any word to its stem. In English, the most famous stemming algorithm is Martin Porter’s stemming algorithm. The Porter stemmer<sup>20</sup> is a simplified version of Lovin’s technique that uses a reduced set of about 60 rules (from 260 suffix patterns in Lovin’s technique) and organizes them into sets; conflicts within one subset of rules are resolved before going on to the next. Using stemming for preprocessing data results in a decrease in the size of the indexing structure and an increase in recall, possibly at the cost of precision.

### 27.4.3 Utilizing a Thesaurus

A **thesaurus** comprises a precompiled list of important concepts and the main word that describes each concept for a particular domain of knowledge. For each concept in this list, a set of synonyms and related words is also compiled.<sup>21</sup> Thus, a synonym can be converted to its matching concept during preprocessing. This preprocessing step assists in providing a standard vocabulary for indexing and searching. Usage of a thesaurus, also known as a *collection of synonyms*, has a substantial impact on the recall of information systems. This process can be complicated because many words have different meanings in different contexts.

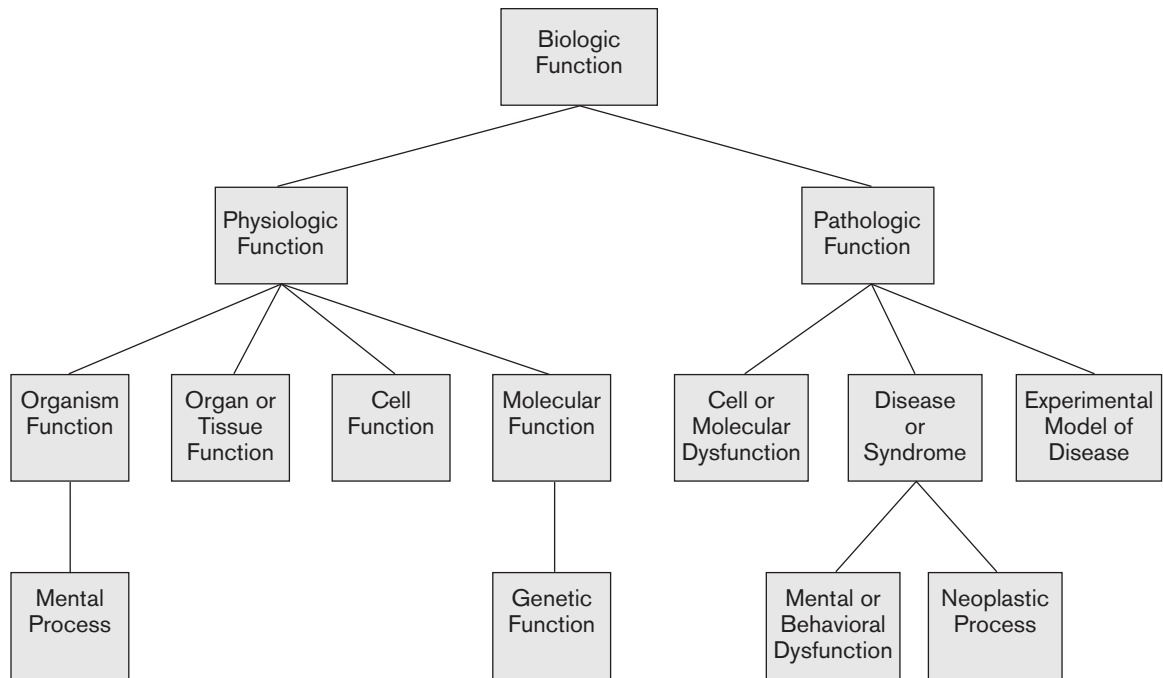
**UMLS**<sup>22</sup> is a large biomedical thesaurus of millions of concepts (called the *meta-thesaurus*) and a semantic network of meta concepts and relationships that organize the metathesaurus (see Figure 27.3). The concepts are assigned labels from the semantic network. This thesaurus of concepts contains synonyms of medical terms, hierarchies of broader and narrower terms, and other relationships among words and concepts that make it a very extensive resource for information retrieval of documents in the medical domain. Figure 27.3 illustrates part of the UMLS Semantic Network.

---

<sup>20</sup>See Porter (1980).

<sup>21</sup>See Baeza-Yates and Ribeiro-Neto (1999).

<sup>22</sup>Unified Medical Language System from the National Library of Medicine.

**Figure 27.3**

A portion of the UMLS Semantic Network: “Biologic Function” Hierarchy.

Source: UMLS Reference Manual, National Library of Medicine.

**WordNet**<sup>23</sup> is a manually constructed thesaurus that groups words into strict synonym sets called *synsets*. These synsets are divided into noun, verb, adjective, and adverb categories. Within each category, these synsets are linked together by appropriate relationships such as class/subclass or “is-a” relationships for nouns.

WordNet is based on the idea of using a controlled vocabulary for indexing, thereby eliminating redundancies. It is also useful in providing assistance to users with locating terms for proper query formulation.

#### 27.4.4 Other Preprocessing Steps: Digits, Hyphens, Punctuation Marks, Cases

Digits, dates, phone numbers, e-mail addresses, URLs, and other standard types of text may or may not be removed during preprocessing. Web search engines, however, index them in order to use this type of information in the document metadata to improve precision and recall (see Section 27.6 for detailed definitions of *precision* and *recall*).

<sup>23</sup>See Fellbaum (1998) for a detailed description of WordNet.

Hyphens and punctuation marks may be handled in different ways. Either the entire phrase with the hyphens/punctuation marks may be used, or they may be eliminated. In some systems, the character representing the hyphen/punctuation mark may be removed, or may be replaced with a space. Different information retrieval systems follow different rules of processing. Handling hyphens automatically can be complex: it can either be done as a classification problem, or more commonly by some heuristic rules. For example, the `StandardTokenizer` in Lucene<sup>24</sup> treats the hyphen as a delimiter to break words—with the exception that if there is a number in the token, the words are not split (for example, words like AK-47, phone numbers, etc.). Many domain-specific terms like product catalogs, different versions of a product, and so on have hyphens in them. When search engines crawl the Web for indexing, it becomes difficult to automatically treat hyphens correctly; therefore, simpler strategies are devised to process hyphens.

Most information retrieval systems perform case-insensitive search, converting all the letters of the text to uppercase or lowercase. It is also worth noting that many of these text preprocessing steps are language specific, such as involving accents and diacritics and the idiosyncrasies that are associated with a particular language.

### 27.4.5 Information Extraction

**Information extraction** (IE) is a generic term used for extracting structured content from text. Text analytic tasks such as identifying noun phrases, facts, events, people, places, and relationships are examples of IE tasks. These tasks are also called *named entity recognition tasks* and use rule-based approaches with either a thesaurus, regular expressions and grammars, or probabilistic approaches. For IR and search applications, IE technologies are mostly used to identify named entities that involve text analysis, matching, and categorization for improving the relevance of search systems. Language technologies using part-of-speech tagging are applied to semantically annotate the documents with extracted features to aid search relevance.

## 27.5 Inverted Indexing

The simplest way to search for occurrences of query terms in text collections can be performed by sequentially scanning the text. This kind of online searching is only appropriate when text collections are small. Most information retrieval systems process the text collections to create indexes and operate upon the inverted index data structure (refer to the indexing task in Figure 27.1). An inverted index structure comprises vocabulary and document information. **Vocabulary** is a set of distinct query terms in the document set. Each term in a vocabulary set has an associated collection of information about the documents that contain the term, such as document id, occurrence count, and offsets within the document where the

<sup>24</sup>See further details on `StandardTokenizer` at <https://lucene.apache.org/>

term occurs. The simplest form of vocabulary terms consists of words or individual tokens of the documents. In some cases, these vocabulary terms also consist of phrases,  $n$ -grams, entities, links, names, dates, or manually assigned descriptor terms from documents and/or Web pages. For each term in the vocabulary, the corresponding document id's, occurrence locations of the term in each document, number of occurrences of the term in each document, and other relevant information may be stored in the document information section.

Weights are assigned to document terms to represent an estimate of the usefulness of the given term as a descriptor for distinguishing the given document from other documents in the same collection. A term may be a better descriptor of one document than of another by the weighting process (see Section 27.2).

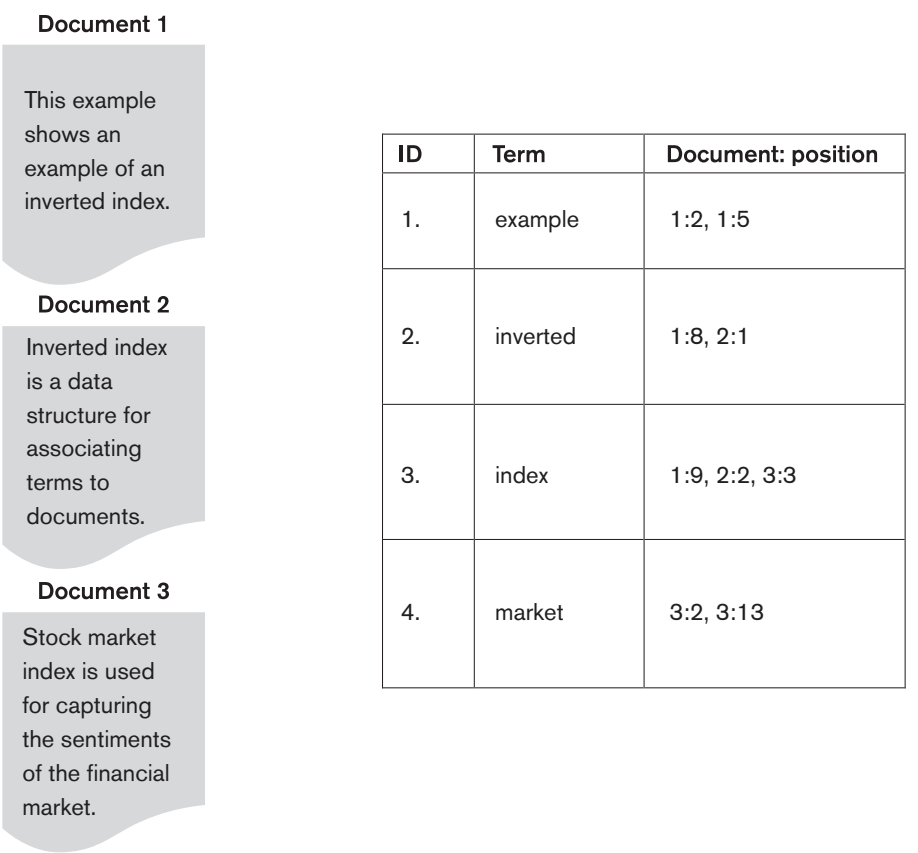
An **inverted index** of a document collection is a data structure that attaches distinct terms with a list of all documents that contains the term. The process of inverted index construction involves the extraction and processing steps shown in Figure 27.2. Acquired text is first preprocessed and the documents are represented with the vocabulary terms. Documents' statistics are collected in document lookup tables. Statistics generally include counts of vocabulary terms in individual documents as well as different collections, their positions of occurrence within the documents, and the lengths of the documents. The vocabulary terms are weighted at indexing time according to different criteria for collections. For example, in some cases terms in the titles of the documents may be weighted more heavily than terms that occur in other parts of the documents.

One of the most popular weighting schemes is the TF-IDF (term frequency-inverse document frequency) metric that we described in Section 27.2. For a given term, this weighting scheme distinguishes to some extent the documents in which the term occurs more often from those in which the term occurs very little or never. These weights are normalized to account for varying document lengths, further ensuring that longer documents with proportionately more occurrences of a word are not favored for retrieval over shorter documents with proportionately fewer occurrences. These processed document-term streams (matrices) are then inverted into term-document streams (matrices) for further IR steps.

Figure 27.4 shows an illustration of term-document-position vectors for the four illustrative terms—*example*, *inverted*, *index*, and *market*—which shows the positions where each term occurs in the three documents.

The steps involved in inverted index construction can be summarized as follows:

1. Break the documents into vocabulary terms by tokenizing, cleansing, removing stopwords, stemming, and/or using an additional thesaurus as vocabulary.
2. Collect document statistics and store the statistics in a document lookup table.
3. Invert the document-term stream into a term-document stream along with additional information such as term frequencies, term positions, and term weights.



**Figure 27.4**  
Example of an inverted index.

Searching for relevant documents from the inverted index, given a set of query terms, is generally a three-step process.

- 1. Vocabulary search.** If the query comprises multiple terms, they are separated and treated as independent terms. Each term is searched in the vocabulary. Various data structures, like variations of B<sup>+</sup>-tree or hashing, may be used to optimize the search process. Query terms may also be ordered in lexicographic order to improve space efficiency.
- 2. Document information retrieval.** The document information for each term is retrieved.
- 3. Manipulation of retrieved information.** The document information vector for each term obtained in step 2 is now processed further to incorporate various forms of query logic. Various kinds of queries like prefix, range, context, and proximity queries are processed in this step to construct the final result based on the document collections returned in step 2.

### 27.5.1 Introduction to Lucene

Lucene is an actively maintained open source indexing/search engine that has become popular in both academic and commercial settings. Indexing is the primary focus of Lucene, but it uses indexing to facilitate search. The Lucene library is written in Java and comes with out-of-the-box scalable and high-performance capability. Lucene is the engine that powers another widely popular enterprise search application called Solr.<sup>25</sup> Solr provides many add-on capabilities to Lucene, such as providing Web interfaces for indexing many different document formats.

An upcoming book by Moczar (2015) discusses both Lucene and Solr.

**Indexing:** In Lucene, documents must go through a process of indexing before they become available for search. A Lucene document is made up of a set of fields. Fields hold the type of data in the index and are loosely comparable to columns in a database table. A field can be of type binary, numeric, or text data. Text fields consist of either entire chunk of untokenized text or a series of processed lexical units called token streams. The token streams are created via application of different types of available tokenization and filtering algorithms. For example, StandardTokenizer is one of the available tokenizers in Lucene that implements Unicode text segmentation for splitting words apart. There are other tokenizers, such as a WhitespaceTokenizer, that divide text at whitespaces. It is also easy to extend these tokenizers and filters in Lucene to create custom text analysis algorithms for tokenization and filtering. These analysis algorithms are central to achieving desired search results. Lucene provides APIs and several implementations for many high-speed and efficient tokenization and filtering algorithms. These algorithms have been extended for several different languages and domains, and they feature implementations of natural language processing algorithms for stemming, conducting dictionary-driven lemmatization, performing morphological analysis, conducting phonetic analysis, and so on.

**Search:** With a powerful search API, queries are matched against documents and a ranked list of results is retrieved. Queries are compared against the term vectors in inverted indexes to compute relevance scores based on the vector space model (see Section 27.2.2). Lucene provides a highly configurable search API wherein one can create queries for wildcard, exact, Boolean, proximity, and range searches. Lucene's default scoring algorithm uses variants of TF-IDF scoring to rank search results. To speed up search, Lucene maintains document-dependent normalization factors precomputed at index time; these are called norms of term vectors in document fields. These precomputed norms speed up the scoring process in Lucene. The actual query matching algorithms use functions that do very little computation at query matching time.

**Applications:** One of the reasons for Lucene's immense popularity is the ease of availability of Lucene applications for handling various document collections and

---

<sup>25</sup>See <http://lucene.apache.org/solr/>

deployment systems for indexing large unstructured document collections. The enterprise search application built on top of Lucene is called Solr. Solr is a Web server application that provides support for faceted search (see Section 27.8.1 on faceted search), custom format document processing support (such as PDF, HTML, etc.), and Web services for several API functions for indexing and search in Lucene.

## 27.6 Evaluation Measures of Search Relevance

Without proper evaluation techniques, one cannot compare and measure the relevance of different retrieval models and IR systems in order to make improvements. Evaluation techniques of IR systems measure the *topical relevance* and *user relevance*. **Topical relevance** measures the extent to which the topic of a result matches the topic of the query. Mapping one's information need with "perfect" queries is a cognitive task, and many users are not able to effectively form queries that would retrieve results more suited to their information need. Also, since a major chunk of user queries are informational in nature, there is no fixed set of right answers to show to the user. **User relevance** is a term used to describe the "goodness" of a retrieved result with regard to the user's information need. User relevance includes other implicit factors, such as user perception, context, timeliness, the user's environment, and current task needs. Evaluating user relevance may also involve subjective analysis and study of user retrieval tasks to capture some of the properties of implicit factors involved in accounting for users' bias for judging performance.





In Web information retrieval, no binary classification decision is made on whether a document is relevant or nonrelevant to a query (whereas the Boolean (or binary) retrieval model uses this scheme, as we discussed in Section 27.2.1). Instead, a ranking of the documents is produced for the user. Therefore, some evaluation measures focus on comparing different rankings produced by IR systems. We discuss some of these measures next.

### 27.6.1 Recall and Precision

Recall and precision metrics are based on the binary relevance assumption (whether each document is relevant or nonrelevant to the query). **Recall** is defined as the number of relevant documents retrieved by a search divided by the total number of actually relevant documents existing in the database. **Precision** is defined as the number of relevant documents retrieved by a search divided by the total number of documents retrieved by that search. Figure 27.5 is a pictorial representation of the terms *retrieved* versus *relevant* and shows how search results relate to four different sets of documents.

The notation for Figure 27.5 is as follows:

- TP: true positive
- FP: false positive

		Relevant?	
		Yes	No
Retrieved?	Yes	 Hits TP	 False Alarms FP
	No	Misses  FN	Correct Rejections TN 

**Figure 27.5**

Retrieved versus relevant search results.

- FN: false negative
- TN: true negative

The terms *true positive*, *false positive*, *false negative*, and *true negative* are generally used in any type of classification tasks to compare the given classification of an item with the desired correct classification. Using the term *hits* for the documents that truly or “correctly” match the user request, we can define *recall* and *precision* as follows:

$$\text{Recall} = |\text{Hits}|/|\text{Relevant}|$$

$$\text{Precision} = |\text{Hits}|/|\text{Retrieved}|$$

Recall and precision can also be defined in a ranked retrieval setting. Let us assume that there is one document at each rank position. The recall at rank position  $i$  for document  $d_i^q$  (denoted by  $r(i)$ ) ( $d_i^q$  is the retrieved document at position  $i$  for query  $q$ ) is the fraction of relevant documents from  $d_1^q$  to  $d_i^q$  in the result set for the query. Let the set of relevant documents from  $d_1^q$  to  $d_i^q$  in that set be  $S_i$  with cardinality  $|S_i|$ . Let  $(|D_q|)$  be the size of relevant documents for the query. In this case,  $|S_i| \leq |D_q|$ . Then:

$$\text{Ranked\_retrieval\_recall: } r(i) = |S_i|/|D_q|$$

The precision at rank position  $i$  or document  $d_i^q$  (denoted by  $p(i)$ ) is the fraction of documents from  $d_1^q$  to  $d_i^q$  in the result set that are relevant:

$$\text{Ranked\_retrieval\_precision: } p(i) = |S_i|/i$$

Table 27.2 illustrates the  $p(i)$ ,  $r(i)$ , and average precision (discussed in the next section) metrics. It can be seen that recall can be increased by presenting more results to the user, but this approach runs the risk of decreasing the precision. In the example, the number of relevant documents for some query = 10. The rank position and the relevance of an individual document are shown. The precision and recall value can be computed at each position within the ranked list as shown in the last two columns. As we see in Table 27.2, the ranked\_retrieval\_recall rises monotonically whereas the precision is prone to fluctuation.



**Table 27.2** Precision and Recall for Ranked Retrieval

Doc. No.	Rank Position $i$	Relevant	Precision( $i$ )	Recall( $i$ )
10	1	Yes	1/1 = 100%	1/10 = 10%
2	2	Yes	2/2 = 100%	2/10 = 20%
3	3	Yes	3/3 = 100%	3/10 = 30%
5	4	No	3/4 = 75%	3/10 = 30%
17	5	No	3/5 = 60%	3/10 = 30%
34	6	No	3/6 = 50%	3/10 = 30%
215	7	Yes	4/7 = 57.1%	4/10 = 40%
33	8	Yes	5/8 = 62.5%	5/10 = 50%
45	9	No	5/9 = 55.5%	5/10 = 50%
16	10	Yes	6/10 = 60%	6/10 = 60%

### 27.6.2 Average Precision

Average precision is computed based on the precision at each relevant document in the ranking. This measure is useful for computing a single precision value to compare different retrieval algorithms on a query  $q$ .

$$P_{\text{avg}} = \sum_{d_i^r \in D_q} p(i) / |D_q|$$

Consider the sample precision values of relevant documents in Table 27.2. The average precision ( $P_{\text{avg}}$  value) for the example in Table 27.2 is  $P(1) + P(2) + P(3) + P(7) + P(8) + P(10)/6 = 79.93\%$  (only relevant documents are considered in this calculation). Many good algorithms tend to have high top- $k$  average precision for small values of  $k$ , with correspondingly low values of recall.

### 27.6.3 Recall/Precision Curve

A recall/precision curve can be drawn based on the recall and precision values at each rank position, where the  $x$ -axis is the recall and the  $y$ -axis is the precision. Instead of using the precision and recall at each rank position, the curve is commonly plotted using recall levels  $r(i)$  at 0%, 10%, 20% ... 100%. The curve usually has a negative slope, reflecting the inverse relationship between precision and recall.

### 27.6.4 F-Score

$F$ -score ( $F$ ) is the harmonic mean of the precision ( $p$ ) and recall ( $r$ ) values. That is,

$$\frac{1}{F} = \frac{\frac{1}{p} + \frac{1}{r}}{2}$$

High precision is achieved almost always at the expense of recall and vice versa. It is a matter of the application's context whether to tune the system for high precision or high recall. *F*-score is typically used as a single measure that combines precision and recall to compare different result sets:

$$F = \frac{2pr}{p+r}$$

One of the properties of harmonic mean is that the harmonic mean of two numbers tends to be closer to the smaller of the two. Thus *F* is automatically biased toward the smaller of the precision and recall values. Therefore, for a high *F*-score, both precision and recall must be high.

$$F = \frac{2}{\frac{1}{p} + \frac{1}{r}}$$

## 27.7 Web Search and Analysis<sup>26</sup>

The emergence of the Web has brought millions of users to search for information, which is stored in a very large number of active sites. To make this information accessible, search engines such as Google, Bing and Yahoo! must crawl and index these sites and document collections in their index databases. Moreover, search engines must regularly update their indexes given the dynamic nature of the Web as new Web sites are created and current ones are updated or deleted. Since there are many millions of pages available on the Web on different topics, search engines must apply many sophisticated techniques such as link analysis to identify the importance of pages.

There are other types of search engines besides the ones that regularly crawl the Web and create automatic indexes: these are human-powered, vertical search engines or metasearch engines. These search engines are developed with the help of computer-assisted systems to aid the curators with the process of assigning indexes. They consist of manually created specialized Web directories that are hierarchically organized indexes to guide user navigation to different resources on the Web. **Vertical search engines** are customized topic-specific search engines that crawl and index a specific collection of documents on the Web and provide search results from that specific collection. **Metasearch engines** are built on top of search engines: they query different search engines simultaneously and aggregate and provide search results from these sources.

Another source of searchable Web documents is digital libraries. **Digital libraries** can be broadly defined as collections of electronic resources and services for the delivery of materials in a variety of formats. These collections may include a university's library catalog, catalogs from a group of participating universities, as in the

---

<sup>26</sup>The contribution of Pranesh P. Ranganathan and Hari P. Kumar to this section is appreciated.

State of Florida University System, or a compilation of multiple external resources on the World Wide Web, such as Google Scholar or the IEEE/ACM index. These interfaces provide universal access to different types of content—such as books, articles, audio, and video—situated in different database systems and remote repositories. Similar to real libraries, these digital collections are maintained via a catalog and organized in categories for online reference. Digital libraries “include personal, distributed, and centralized collections such as online public-access catalogs (OPACs) and bibliographic databases, distributed document databases, scholarly and professional discussion lists and electronic journals, other online databases, forums, and bulletin boards.”<sup>27</sup>

### 27.7.1 Web Analysis and Its Relationship to Information Retrieval

In addition to browsing and searching the Web, another important activity closely related to information retrieval is to *analyze* or *mine* information on the Web for new information of interest. (We discuss mining of data from files and databases in Chapter 28.) Application of data analysis techniques for discovery and analysis of useful information from the Web is known as **Web analysis**. Over the past few years, the World Wide Web has emerged as an important repository of information for many day-to-day applications for individual consumers, as well as a significant platform for e-commerce and for social networking. These properties make it an interesting target for data analysis applications. The Web mining and analysis field is an integration of a wide range of fields spanning information retrieval, text analysis, natural language processing, data mining, machine learning, and statistical analysis.

The goals of Web analysis are to improve and personalize search results relevance and to identify trends that may be of value to various businesses and organizations. We elaborate on these goals next.

- **Finding relevant information.** People usually search for specific information on the Web by entering keywords in a search engine or browsing information portals and using services. Search services are heavily constrained by search relevance problems since search engines must map and approximate the information need of millions of users as an *a priori* task. Low *precision* (see Section 27.6) ensues due to results that are nonrelevant to the user. In the case of the Web, high *recall* (see Section 27.6) is impossible to determine due to the inability to index all the pages on the Web. Also, measuring recall does not make sense since the user is concerned with only the top few documents. The most relevant results for the user are typically from only the top few results.
- **Personalization of the information.** Different people have different content and presentation preferences. Various customization tools used in

---

<sup>27</sup>Covi and Kling (1996), page 672.

Web-based applications and services (such as click-through monitoring, eyeball tracking, explicit or implicit user profile learning, and dynamic service composition using Web APIs) are used for service adaptation and personalization. A personalization engine typically has algorithms that make use of the user's personalization information—collected by various tools—to generate user-specific search results. The Web has become a rich landscape where people leave traces as they navigate, click, like, comment, and buy things in this virtual space. This information is of high commercial value, and many companies in all kinds of consumer goods mine and sell this information for customer targeting.

- **Finding information of social value.** With more than 1 billion downloads of the Facebook app on various Android devices, one can imagine how popular the various social networks have become in recent times. People build what is called social capital in these virtual worlds such as Twitter and Facebook. **Social capital** refers to features of social organizations, such as networks, norms, and social trust, that facilitate coordination and cooperation for mutual benefit. Social scientists are studying social capital and how to harness this rich resource to benefit society in various ways. We briefly touch upon aspects of social search in Section 27.8.2.

Web analysis can be further classified into three categories: **Web structure analysis**, which discovers knowledge from hyperlinks that represent the structure of the Web; **Web content analysis**, which deals with extracting useful information/knowledge from Web page contents; and **Web usage analysis**, which mines user access patterns from usage logs that record the activity of every user.

### 27.7.2 Web Structure Analysis

The World Wide Web is a huge corpus of information, but locating resources that are both high quality and relevant to the needs of the user is very difficult. The set of Web pages taken as a whole has almost no unifying structure, with variability in authoring style and content; this variability makes it difficult to precisely locate needed information. Index-based search engines have been one of the primary tools by which users search for information on the Web. Web search engines **crawl** the Web and create an index to the Web for searching purposes. When a user specifies her need for information by supplying keywords, these Web search engines query their repository of indexes and produce links or URLs with abbreviated content as search results. There may be thousands of pages relevant to a particular query. A problem arises when only a few most relevant results are returned to the user. Our discussions of querying and relevance-based ranking in IR systems in (see Sections 27.2 and 27.3) is applicable to Web search engines. These ranking algorithms explore the link structure of the Web.

Web pages, unlike standard text collections, contain connections to other Web pages or documents (via the use of hyperlinks), allowing users to browse from page to page. A **hyperlink** has two components: a **destination page** and an **anchor text** that describes the link. For example, a person can link to the Yahoo Web site on her

Web page with anchor text such as “My favorite Web site.” Anchor texts can be thought of as being implicit endorsements. They provide important latent human annotation. A person linking to other Web pages from her Web page is assumed to have some relation to those Web pages. Web search engines aim to distill results per their relevance and authority. There are many redundant hyperlinks, like the links to the homepage on every Web page of the Web site. Such hyperlinks must be eliminated from the search results by the search engines.

A **hub** is a Web page or a Web site that links to a collection of prominent sites (authorities) on a common topic. A good **authority** is a page that is pointed to by many good hubs, whereas a good hub is a page that points to many good authorities. These ideas are used by the HITS ranking algorithm. We briefly discuss a couple of ranking algorithms in the next section.

### 27.7.3 Analyzing the Link Structure of Web Pages

The goal of **Web structure analysis** is to generate a structural representation about the Web site and Web pages. Web structure analysis focuses on the inner structure of documents and deals with the link structure using hyperlinks at the interdocument level. The structure and content of Web pages are often combined for information retrieval by Web search engines. Given a collection of interconnected Web documents, interesting and informative facts describing their connectivity in the Web subset can be discovered. Web structure analysis is also used to help with navigation and make it possible to compare/integrate different Web page schemes. This aspect of Web structure analysis facilitates Web document classification and clustering on the basis of structure.

**The PageRank Ranking Algorithm.** As discussed earlier, ranking algorithms are used to order search results based on relevance and authority. Google uses the well-known **PageRank** algorithm,<sup>28</sup> which is based on the “importance” of each page. Every Web page has a number of forward links (out-edges) and backlinks (in-edges). It is very difficult to determine all the backlinks of a Web page, whereas it is relatively straightforward to determine its forward links. According to the PageRank algorithm, highly linked pages are more important (have greater authority) than pages with fewer links. However, not all backlinks are important. A backlink to a page from a credible source is more important than a link from some arbitrary page. Thus a page has a high rank if the sum of the ranks of its backlinks is high. PageRank was an attempt to see how good an approximation of the “importance” of a page can be obtained from the link structure.

The computation of page ranking follows an iterative approach. PageRank of a Web page is calculated as a sum of the PageRanks of all its backlinks. PageRank treats the Web like a *Markov model*. An imaginary Web surfer visits an infinite string of pages by clicking randomly. The PageRank of a page is an estimate of how often the surfer winds

---

<sup>28</sup>The PageRank algorithm was proposed by Lawrence Page (1998) and Sergey Brin, founders of Google. For more information, see <http://en.wikipedia.org/wiki/PageRank>

up at a particular page. PageRank is a measure of the query-independent importance of a page/node. For example, let  $P(X)$  be the PageRank of any page  $X$  and  $C(X)$  be the number of outgoing links from page  $X$ , and let  $d$  be the damping factor in the range  $0 < d < 1$ . Usually  $d$  is set to 0.85. Then PageRank for a page  $A$  can be calculated as:

$$P(A) = (1 - d) + d(P(T_1)/C(T_1) + P(T_2)/C(T_2) + \dots + P(T_n)/C(T_n))$$

Here  $T_1, T_2, \dots, T_n$  are the pages that point to Page  $A$  (that is, are citations to page  $A$ ). PageRank forms a probability distribution over Web pages, so the sum of all Web pages' PageRanks is one.

**The HITS Ranking Algorithm.** The HITS<sup>29</sup> algorithm proposed by Jon Kleinberg is another type of ranking algorithm exploiting the link structure of the Web. The algorithm presumes that a good hub is a document that points to many hubs, and a good authority is a document that is pointed at by many other authorities. The algorithm contains two main steps: a sampling component and a weight-propagation component. The sampling component constructs a focused collection  $S$  of pages with the following properties:

1.  $S$  is relatively small.
2.  $S$  is rich in relevant pages.
3.  $S$  contains most (or a majority) of the strongest authorities.

The weight component recursively calculates the hub and authority values for each document as follows:

1. Initialize hub and authority values for all pages in  $S$  by setting them to 1.
2. While (hub and authority values do not converge):
  - a. For each page in  $S$ , calculate authority value = Sum of hub values of all pages *pointing to* the current page.
  - b. For each page in  $S$ , calculate hub value = Sum of authority values of all pages *pointed at by* the current page.
  - c. Normalize hub and authority values such that sum of all hub values in  $S$  equals 1 and the sum of all authority values in  $S$  equals 1.

## 27.7.4 Web Content Analysis

As mentioned earlier, **Web content analysis** refers to the process of discovering useful information from Web content/data/documents. The **Web content data** consists of unstructured data such as free text from electronically stored documents, semi-structured data typically found as HTML documents with embedded image data, and more structured data such as tabular data and pages in HTML, XML, or other markup languages generated as output from databases. More generally, the term *Web content* refers to any real data in the Web page that is intended for the user accessing that page. This usually consists of but is not limited to text and graphics.

---

<sup>29</sup>See Kleinberg (1999).

We will first discuss some preliminary Web content analysis tasks and then look at the traditional analysis tasks of Web page classification and clustering.

**Structured Data Extraction.** Structured data on the Web is often very important because it represents essential information, such as a structured table showing the airline flight schedule between two cities. There are several approaches to structured data extraction. One includes writing a **wrapper**, or a program that looks for different structural characteristics of the information on the page and extracts the right content. Another approach is to manually write an extraction program for each Web site based on observed format patterns of the site, which is very labor intensive and time consuming. This latter approach does not scale to a large number of sites. A third approach is **wrapper induction** or **wrapper learning**, where the user first manually labels a set of training set pages and the learning system generates rules—based on the learning pages—that are applied to extract target items from other Web pages. A fourth approach is the automatic approach, which aims to find patterns/grammars from the Web pages and then uses **wrapper generation** to produce a wrapper to extract data automatically.

**Web Information Integration.** The Web is immense and has billions of documents, authored by many different persons and organizations. Because of this, Web pages that contain similar information may have different syntax and different words that describe the same concepts. This creates the need for integrating information from diverse Web pages. Two popular approaches for Web information integration are:

1. **Web query interface integration**, to enable querying multiple Web databases that are not visible in external interfaces and are hidden in the “deep Web.” The **deep Web**<sup>30</sup> consists of those pages that do not exist until they are created dynamically as the result of a specific database search, which produces some of the information in the page (see Chapter 11). Since traditional search engine crawlers cannot probe and collect information from such pages, the deep Web has heretofore been hidden from crawlers.
2. **Schema matching**, such as integrating directories and catalogs to come up with a global schema for applications. An example of such an application would be to match and combine into one record data from various sources by cross-linking health records from multiple systems. The result would be an individual global health record.

These approaches remain an area of active research, and a detailed discussion of them is beyond the scope of this text. Consult the Selected Bibliography at the end of this chapter for further details.

**Ontology-Based Information Integration.** This task involves using ontologies to effectively combine information from multiple heterogeneous sources. Ontologies—formal models of representation with explicitly defined concepts and named

---

<sup>30</sup>The deep Web as defined by Bergman (2001).



relationships linking them—are used to address the issues of semantic heterogeneity in data sources. Different classes of approaches are used for information integration using ontologies.

- **Single ontology approaches** use one global ontology that provides a shared vocabulary for the specification of the semantics. They work if all information sources to be integrated provide nearly the same view on a domain of knowledge. For example, UMLS (described in Section 27.4.3) can serve as a common ontology for biomedical applications.
- In a **multiple ontology approach**, each information source is described by its own ontology. In principle, the “source ontology” can be a combination of several other ontologies, but it cannot be assumed that the different “source ontologies” share the same vocabulary. Dealing with multiple, partially overlapping, and potentially conflicting ontologies is a difficult problem faced by many applications, including those in bioinformatics and other complex topics of study.

**Building Concept Hierarchies.** One common way of organizing search results is via a linear ranked list of documents. But for some users and applications, a better way to display results would be to create groupings of related documents in the search result. One way of organizing documents in a search result, and for organizing information in general, is by creating a **concept hierarchy**. The documents in a search result are organized into groups in a hierarchical fashion. Other related techniques to organize documents are through **classification** and **clustering** (see Chapter 28). Clustering creates groups of documents, where the documents in each group share many common concepts.

**Segmenting Web Pages and Detecting Noise.** There are many superfluous parts in a Web document, such as advertisements and navigation panels. The information and text in these superfluous parts should be eliminated as noise before classifying the documents based on their content. Hence, before applying classification or clustering algorithms to a set of documents, the areas or blocks of the documents that contain noise should be removed.

### 27.7.5 Approaches to Web Content Analysis

The two main approaches to Web content analysis are (1) agent based (IR view) and (2) database based (DB view).

The **agent-based approach** involves the development of sophisticated artificial intelligence systems that can act autonomously or semi-autonomously on behalf of a particular user, to discover and process Web-based information. Generally, the agent-based Web analysis systems can be placed into the following three categories:

- **Intelligent Web agents** are software agents that search for relevant information using characteristics of a particular application domain (and possibly a user profile) to organize and interpret the discovered information. For



example, an intelligent agent retrieves product information from a variety of vendor sites using only general information about the product domain.

- **Information filtering/categorization** is another technique that utilizes Web agents for categorizing Web documents. These Web agents use methods from information retrieval, as well as semantic information based on the links among various documents, to organize documents into a concept hierarchy.
- **Personalized Web agents** are another type of Web agents that utilize the personal preferences of users to organize search results, or to discover information and documents that could be of value for a particular user. User preferences could be learned from previous user choices, or from other individuals who are considered to have similar preferences to the user.

The **database-based approach** aims to infer the structure of the Web site or to transform a Web site to organize it as a database so that better information management and querying on the Web become possible. This approach of Web content analysis primarily tries to model the data on the Web and integrate it so that more sophisticated queries than keyword-based search can be performed. These could be achieved by finding the schema of Web documents or building a Web document warehouse, a Web knowledge base, or a virtual database. The database-based approach may use a model such as the Object Exchange Model (OEM),<sup>31</sup> which represents semistructured data by a labeled graph. The data in the OEM is viewed as a graph, with objects as the vertices and labels on the edges. Each object is identified by an object identifier and a value that is either atomic—such as integer, string, GIF image, or HTML document—or complex in the form of a set of object references.

The main focus of the database-based approach has been with the use of multilevel databases and Web query systems. A **multilevel database** at its lowest level is a database containing primitive semistructured information stored in various Web repositories, such as hypertext documents. At the higher levels, metadata or generalizations are extracted from lower levels and organized in structured collections such as relational or object-oriented databases. In a **Web query system**, information about the content and structure of Web documents is extracted and organized using database-like techniques. Query languages similar to SQL can then be used to search and query Web documents. These types of queries combine structural queries, based on the organization of hypertext documents, and content-based queries.

### 27.7.6 Web Usage Analysis

**Web usage analysis** is the application of data analysis techniques to discover usage patterns from Web data, in order to understand and better serve the needs of Web-based applications. This activity does not directly contribute to information retrieval; but it is important for improving and enhancing users' search experiences.

---

<sup>31</sup>See Kosala and Blockeel (2000).

**Web usage data** describes the pattern of usage of Web pages, such as IP addresses, page references, and the date and time of accesses for a user, user group, or an application. Web usage analysis typically consists of three main phases: preprocessing, pattern discovery, and pattern analysis.

1. **Preprocessing.** Preprocessing converts the information collected about usage statistics and patterns into a form that can be utilized by the pattern discovery methods. For example, we use the term *page view* to refer to pages viewed or visited by a user. There are several different types of preprocessing techniques available:
  - **Usage preprocessing** analyzes the available collected data about usage patterns of users, applications, and groups of users. Because this data is often incomplete, the process is difficult. Data cleaning techniques are necessary to eliminate the impact of irrelevant items in the analysis result. Frequently, usage data is identified by an IP address and consists of clicking streams that are collected at the server. Better data is available if a usage tracking process is installed at the client site.
  - **Content preprocessing** is the process of converting text, image, scripts, and other content into a form that can be used by the usage analysis. Often, this process consists of performing content analysis such as classification or clustering. The clustering or classification techniques can group usage information for similar types of Web pages, so that usage patterns can be discovered for specific classes of Web pages that describe particular topics. Page views can also be classified according to their intended use, such as for sales or for discovery or for other uses.
  - **Structure preprocessing** can be done by parsing and reformatting the information about hyperlinks and structure between viewed pages. One difficulty is that the site structure may be dynamic and may have to be constructed for each server session.
2. **Pattern discovery.** The techniques that are used in pattern discovery are based on methods from the fields of statistics, machine learning, pattern recognition, data analysis, data mining, and other similar areas. These techniques are adapted so they take into consideration the specific knowledge and characteristics of Web analysis. For example, in association rule discovery (see Section 28.2), the notion of a transaction for market-basket analysis considers the items to be unordered. But the order of accessing of Web pages is important, and so it should be considered in Web usage analysis. Hence, pattern discovery involves mining sequences of page views. In general, using Web usage data, the following types of data mining activities may be performed for pattern discovery.
  - **Statistical analysis.** Statistical techniques are the most common method of extracting knowledge about visitors to a Web site. By analyzing the session log, it is possible to apply statistical measures such as mean, median, and frequency count to parameters such as pages viewed, viewing time per page, length of navigation paths between pages, and other parameters that are relevant to Web usage analysis.

- **Association rules.** In the context of Web usage analysis, association rules refer to sets of pages that are accessed together with a support value exceeding some specified threshold. (See Section 28.2 on association rules.) These pages may not be directly connected to one another via hyperlinks. For example, association rule discovery may reveal a correlation between users who visited a page containing electronic products to those who visit a page about sporting equipment.
  - **Clustering.** In the Web usage domain, there are two kinds of interesting clusters to be discovered: usage clusters and page clusters. **Clustering of users** tends to establish groups of users exhibiting similar browsing patterns. Such knowledge is especially useful for inferring user demographics in order to perform market segmentation in e-commerce applications or provide personalized Web content to the users. **Clustering of pages** is based on the content of the pages, and pages with similar contents are grouped together. This type of clustering can be utilized in Internet search engines and in tools that provide assistance to Web browsing.
  - **Classification.** In the Web domain, one goal is to develop a profile of users belonging to a particular class or category. This requires extraction and selection of features that best describe the properties of a given class or category of users. For example, an interesting pattern that may be discovered would be: 60% of users who placed an online order in /Product/Books are in the 18–25 age group and live in rented apartments.
  - **Sequential patterns.** These kinds of patterns identify sequences of Web accesses, which may be used to predict the next set of Web pages to be accessed by a certain class of users. These patterns can be used by marketers to produce targeted advertisements on Web pages. Another type of sequential pattern pertains to which items are typically purchased following the purchase of a particular item. For example, after purchasing a computer, a printer is often purchased.
  - **Dependency modeling.** Dependency modeling aims to determine and model significant dependencies among the various variables in the Web domain. For example, one may be interested in building a model that represents the various stages a visitor undergoes while shopping in an online store; this model would be based on user actions (e.g., being a casual visitor versus being a serious potential buyer).
3. **Pattern analysis.** The final step is to filter out those rules or patterns that are considered to be not of interest based on the discovered patterns. One common technique for pattern analysis is to use a query language such as SQL to detect various patterns and relationships. Another technique involves loading usage data into a data warehouse with ETL tools and performing OLAP operations to view the data along multiple dimensions (see Section 29.3). It is common to use visualization techniques, such as graphing patterns or assigning colors to different values, to highlight patterns or trends in the data.

### 27.7.7 Practical Applications of Web Analysis

**Web Analytics.** The goal of **web analytics** is to understand and optimize the performance of Web usage. This requires collecting, analyzing, and monitoring the performance of Internet usage data. On-site Web analytics measures the performance of a Web site in a commercial context. This data is typically compared against key performance indicators to measure effectiveness or performance of the Web site as a whole, and it can be used to improve a Web site or improve the marketing strategies.

**Web Spamming.** It has become increasingly important for companies and individuals to have their Web sites/Web pages appear in the top search results. To achieve this, it is essential to understand search engine ranking algorithms and to present the information in one's page in such a way that the page is ranked high when the respective keywords are queried. There is a thin line separating legitimate page optimization for business purposes and spamming. **Web spamming** is thus defined as a deliberate activity to promote one's page by manipulating the results returned by the search engines. Web analysis may be used to detect such pages and discard them from search results.

**Web Security.** Web analysis can be used to find interesting usage patterns of Web sites. If any flaw in a Web site has been exploited, it can be inferred using Web analysis, thereby allowing the design of more robust Web sites. For example, the backdoor or information leak of Web servers can be detected by using Web analysis techniques on abnormal Web application log data. Security analysis techniques such as intrusion detection and denial-of-service attacks are based on Web access pattern analysis.

**Web Crawlers.** These are programs that visit Web pages and create copies of all the visited pages so they can be processed by a search engine for indexing the downloaded pages and providing fast searches. Another use of crawlers is to automatically check and maintain Web sites. For example, the HTML code and the links in a Web site can be checked and validated by the crawler. Another unfortunate use of crawlers is to collect e-mail addresses and other personal information from Web pages; the information is subsequently used in sending spam e-mails.

## 27.8 Trends in Information Retrieval

In this section, we review a few concepts that are being considered in recent research work in information retrieval.

### 27.8.1 Faceted Search

Faceted search is a technique that allows for an integrated search and navigation experience by allowing users to explore by filtering available information. This search technique is often used in ecommerce Web sites and applications and

enables users to navigate a multi-dimensional information space. Facets are generally used for handling three or more dimensions of classification. These multiple dimensions of classification allow the **faceted classification scheme** to classify an object in various ways based on different taxonomical criteria. For example, a Web page may be classified in various ways: by content (airlines, music, news, etc.); by use (sales, information, registration, etc.); by location; by language used (HTML, XML, etc.); and in other ways or facets. Hence, the object can be classified in multiple ways based on multiple taxonomies.

A **facet** defines properties or characteristics of a class of objects. The properties should be mutually exclusive and exhaustive. For example, a collection of art objects might be classified using an artist facet (name of artist), an era facet (when the art was created), a type facet (painting, sculpture, mural, etc.), a country of origin facet, a media facet (oil, watercolor, stone, metal, mixed media, etc.), a collection facet (where the art resides), and so on.

Faceted search uses faceted classification, which enables a user to navigate information along multiple paths corresponding to different orderings of the facets. This contrasts with traditional taxonomies, in which the hierarchy of categories is fixed and unchanging. University of California–Berkeley’s Flamenco project<sup>32</sup> is one of the earlier examples of a faceted search system. Most e-commerce sites today, such as Amazon or Expedia, use faceted search in their search interfaces to quickly compare and navigate various aspects related to search criteria.

## 27.8.2 Social Search

The traditional view of Web navigation and browsing assumes that a single user is searching for information. This view contrasts with previous research by library scientists who studied users’ information-seeking habits. This research demonstrated that additional individuals may be valuable information resources during information search by a single user. More recently, research indicates that there is often direct user cooperation during Web-based information search. Some studies report that significant segments of the user population are engaged in explicit collaboration on joint search tasks on the Web. Active collaboration by multiple parties also occurs in certain cases (for example, enterprise settings); at other times, and perhaps for a majority of searches, users often interact with others remotely, asynchronously, and even involuntarily and implicitly.

Socially enabled online information search (social search) is a new phenomenon facilitated by recent Web technologies. **Collaborative social search** involves different ways for active involvement in search-related activities such as co-located search, remote collaboration on search tasks, use of social network for search, use of expertise networks, use of social data mining or collective intelligence to improve the search process, and use of social interactions to facilitate information seeking and sense making. This social search activity may be done synchronously, asynchronously,

---

<sup>32</sup>Yee (2003) describes faceted metadata for image search.

co-located, or in remote shared workspaces. Social psychologists have experimentally validated that the act of social discussions has facilitated cognitive performance. People in social groups can provide solutions (answers to questions), pointers to databases or to other people (meta-knowledge), and validation and legitimization of ideas; in addition, social groups can serve as memory aids and can help with problem reformulation. **Guided participation** is a process in which people co-construct knowledge in concert with peers in their community. Information seeking is mostly a solitary activity on the Web today. Some recent work on collaborative search reports several interesting findings and the potential of this technology for better information access. It is increasingly common for people to use social networks such as Facebook to seek opinions and clarifications on various topics and to read product reviews before making a purchase.

### 27.8.3 Conversational Information Access

**Conversational information access** is an interactive and collaborative information-finding interaction. The participants engage in a natural human-to-human conversation, and intelligent agents listen to the conversation in the background and perform **intent extraction** to provide participants with need-specific information. Agents use direct or subtle interactions with participants via mobile or wearable communication devices. These interactions require technologies like speaker identification, keyword spotting, automatic speech recognition, semantic understanding of conversations, and discourse analysis as a means of providing users with faster and relevant pointers for conversations. Via technologies like those just mentioned, information access is transformed from a solitary activity to a participatory activity. In addition, information access becomes more goal specific as agents use multiple technologies to gather relevant information and as participants provide conversational feedback to agents.

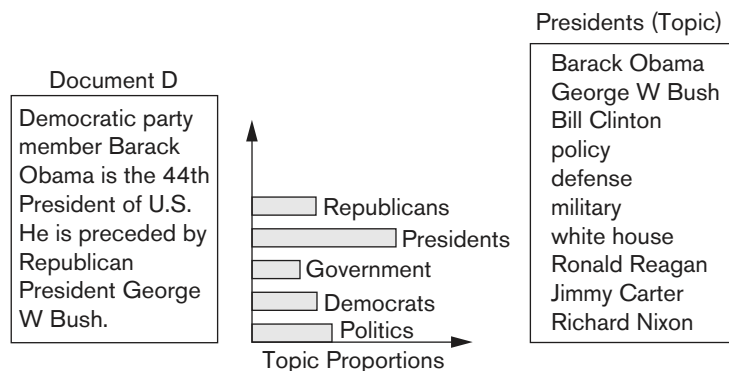
### 27.8.4 Probabilistic Topic Modeling

The unprecedented growth in information generated with the advent of the Web has led to issues concerning how to organize data into categories that will facilitate correct and efficient dissemination of information. For example, international news agencies like Reuters and the Associated Press gather daily news worldwide pertaining to business, sports, politics, technology, and so on. It is a tremendous challenge to organize effectively this plethora of information. Search engines have conventionally organized words within and links among documents to make them accessible on the Web. Organizing information according to the topics and themes of documents allows users to navigate through the vast amount of information based on the topics they are interested in.

To address this problem, a class of machine learning algorithms known as **probabilistic topic models** has emerged in the last decade. These algorithms can automatically organize large collections of documents into relevant themes. The beauty of these algorithms is that they are totally unsupervised, meaning that they

**Figure 27.6**

A document D and its topic proportions.



do not need any training sets or human annotations to perform this thematic extrapolation. The concept of this class of algorithms is as follows: Every document is inherently organized thematically. For example, documents about Barack Obama may mention other presidents, other issues related to the government, or a particular political theme. An article about one of the *Iron Man* movies may contain references to other sci-fi (science fiction) characters from the Marvel series or generally have a sci-fi theme. These inherent structures in documents can be extracted by probabilistic modeling and estimation methods. As another example, let us assume that every document is made up of a collection of different topics in differing proportions (e.g., a document about politics may also be about presidents and American history). Also, every topic is made up of a collection of words.

By considering Figure 27.6, we can guess that document D, which mentions U.S. Presidents Barack Obama and George W. Bush, can belong to the topics Presidents, Politics, Democrats, Republicans, and Government. In general, topics share a fixed vocabulary of words. This vocabulary of words is extracted from the collection of documents for which we wish to train the topic models. We generally choose the number of topics we wish to extract from the collection. Every topic ranks words differently according to how often a word is represented under a certain topic in different documents. In Figure 27.6, the bars representing topic proportions should all sum to 1. Document D primarily belongs to the topic Presidents, as shown in the bar graph. Figure 27.6 depicts the topics related to Presidents along with the list of words associated with this topic.

Probabilistic topic modeling estimates topic distributions using a learning algorithm that assumes that documents can be generated as a mixture of topic proportions. These topic proportion estimates are computed using sampling and expectation maximization algorithms. An algorithm called latent Dirichlet allocation (LDA)<sup>33</sup> is used to generate the topic models. The model assumes a generative process wherein documents are mixtures of latent topics and topics are distributions over words. A generative model randomly generates observable data given

<sup>33</sup>See Blei, Ng, and Jordan (2003).



some hidden parameters. These hidden/unobserved parameters are the Dirichlet distribution<sup>34</sup> priors for words and topics, topic distributions, and per-topic word distributions. Bayesian inference methods such as Gibbs sampling<sup>35</sup> are used to fit the hidden parameters based on the observed data (the words in the documents).

### 27.8.5 Question Answering Systems

Question answering (QA) has become a hot topic of study due to the surge in virtual assistant technology (e.g., Apple's Siri and Microsoft's Cortana). These virtual assistant technologies are advancements in interactive voice response (IVR) systems, which primarily rely on speech recognition techniques such as keyword spotting. Question answering deals with complex understanding of natural language queries. Recently, IBM created history by developing the QA system called Watson, that participated in the *Jeopardy!* Challenge<sup>36</sup> and defeated human players in the popular TV quiz show. Question answering has emerged as a practical engineering discipline that comprises techniques such as parsing; named entity recognition (NER); focus extraction; answer type extraction; relation extraction; ontological inference; and search, indexing, and classification algorithms. Question answering techniques also involve knowledge engineering from large unstructured corpora such as Web document collections and structured databases that incorporate knowledge from various domains. These document collections are generally large enough to require application of big data tools and technologies, some of which we discussed in Chapter 25. In the following sections, we consider the main concepts involved in question answering.

**Types of Questions:** In question answering systems, it is important to know the category or type of question, because answering strategies rely heavily on the type of questions. Some of these categories are not always mutually exclusive and hence require hybrid answering strategies. Generally, questions can be categorized into the following types:

**Factoid Questions:** This type of question pinpoints the right phrase in a document or a database that correctly addresses the question. Examples of this type include questions such as, “Who is the president of the United States?”, “In which city was Elvis Presley born?”, “Where is Hartsfield Jackson International Airport located?”, and “At what time will today's sunset occur?”.

**List Questions:** This type of question seeks a list of factoid responses that satisfy a given criterion. Examples include “Name three plays that were written by Shakespeare”, “Name the male actors who played the role of James Bond in the James Bond 007 movie series”, and “List three red-colored vegetables”.

<sup>34</sup>S. Kotz, N. Balakrishnan, and N. L. Johnson (2000).

<sup>35</sup>German and German (1984).

<sup>36</sup>See Ferrucci et al. (2010).



**Definition Questions:** This type of question asks about the definition and meaning of the concept, and to extract the essential information and properties of the concept. Examples include “What is an inert gas?”, “Who is Alexander the Great?”, and “What is the LIBOR rate?”.

**Opinion Questions:** This type of question seeks different views on a subject that the question. For example, “What countries should be allowed to test nuclear weapons?” and “What is the sentiment in Saudi Arabia about terrorism in the Middle East?”

In recent years, joint initiatives in research and academia have advocated adopting common metrics, architectures, tools, and methodologies to create baselines that will facilitate and improve the QA technique.

**Architectures.** Most state-of-the-art QA architectures are generally made up of pipelines that comprise the following stages:

**Question Analysis:** This stage involves analyzing questions and converting them to structural representations of analyzed text for processing by downstream components. Answer types are extracted from parsed representations of questions using some or all of the following techniques: shallow semantic parsing, focus detection, answer type classification, named entity recognition, and co-reference resolution.

- **Shallow semantic parsing:** The process of assigning surface-level markups to sentence structures via supervised machine learning methods. In general, frames are automatically instantiated for sentences by trying to match “WHO did WHAT to WHOM, WHEN, WHERE, WHY, and HOW” elements.
- **Focus detection:** In an image, certain things stand out whereas others remain in the background. We say that things that stand out are in focus. Similarly, in QA, questions have focus words that contain references to answers. For example, in the question “Which book of Shakespeare is a tragedy about lovers?”, the focus words “book of Shakespeare” can be instantiated with the rule “which X”, where X is a noun phrase in a sentence. QA systems use focus words to trigger directed searches and to aid in answer resolution.
- **Answer type classification:** This phase helps determine the categories of answers in QA. In the preceding example, the headword of the focus words, “book”, is the answer type for this question. Several machine learning techniques are applied in QA to determine the answer type of a question.
- **Named entity recognition:** Named entity recognition seeks to classify elements in text into predefined categories, such as person, place, animal, country, river, continent.
- **Co-reference resolution:** The task of co-reference resolution is about identifying multiple expressions in text that refer to the same thing. For example, in the sentence “John said that he wanted to go to the theater on Sunday”, the pronoun “he” refers to “John” and is a co-reference in text.

**Query Generation:** In this stage, the analyzed text is used to generate multiple queries using query normalization and expansion techniques for one or more underlying search engines in which the answers may be embedded. For example, in the question, “Which book of Shakespeare is about tragedy of lovers?”, the expanded queries can be “Shakespeare love story”, “novels of Shakespeare”, “tragic love story author Shakespeare”, “love story genre tragedy author Shakespeare”, and so on. Extracted keywords, answer types, synonyms information, and named entities are generally used in different combinations to create different queries.

**Search:** In this stage, the queries are sent to different search engines and relevant passages are retrieved. Search engines where searches are performed can be online, such as Google or Bing, and offline, such as Lucene or Indri.<sup>37</sup>

**Candidate Answer Generation:** Named entity extractors are used on retrieved passages and matched against desired answer types to come up with candidate answers. Depending on the desired granularity of the answer, candidate generation and answer type matching algorithms are applied (e.g., surface pattern matching and structural matching). In surface pattern matching, regular expression templates are instantiated with arguments from the question and matched against lexical chunks of retrieved passages to extract answers. For example, focus words are aligned with passages containing potential answers to extract answer candidates. In the sentence, “Romeo and Juliet is a tragic love story by Shakespeare”, the phrase “Romeo and Juliet” can simply replace “Which book” in the question, “Which book is a tragic love story by Shakespeare?”. In structural matching, questions and retrieved passages are parsed and aligned together using syntactic and semantic alignment to find answer candidates. A sentence such as, “Shakespeare wrote the tragic love story Romeo and Juliet” cannot be surface matched with the aforementioned question, but with correct parsing and alignment will structurally match with the question.

**Answer Scoring:** In this stage, confidence scores for the candidate answers are estimated. Similar answers are merged; knowledge sources can be reused to gather supporting evidence for different candidate answers.

## 27.9 Summary

In this chapter, we covered an important area called information retrieval (IR) that is closely related to databases. With the advent of the Web, unstructured data with text, images, audio, and video is proliferating at phenomenal rates. Although database management systems have a very good handle on structured data, the unstructured data containing a variety of data types is being stored mainly on ad hoc information repositories on the Web that are available for consumption primarily via IR systems. Google, Yahoo, and similar search engines are IR systems that make the advances in this field readily available for the average end user and give end users a richer and continually improving search experience.

<sup>37</sup><http://www.lemurproject.org/indri/>

We started in Section 27.1 by first introducing the field of IR in section 27.1.1 and comparing IR and database technologies in Section 27.1.2. A brief history of IR was presented in Section 27.1.3, and the query and browsing modes of interaction in IR systems were introduced in Section 27.1.4.

We presented in Section 27.2 the various retrieval models used in IR, including Boolean, vector space, probabilistic, and semantic models. These models allow us to measure whether a document is relevant to a user query and provide similarity measurement heuristics. In Section 27.3 we presented different types of queries—in addition to keyword-based queries, which dominate, there are other types, including Boolean, phrase, proximity, natural language, and others for which explicit support needs to be provided by the retrieval model. Text preprocessing is important in IR systems, and we discussed in Section 27.4 various activities like stopword removal, stemming, and the use of thesauruses. We then discussed the construction and use of inverted indexes in Section 27.5, which are at the core of IR systems and contribute to factors involving search efficiency. We then discussed in Section 27.6 various evaluation metrics, such as recall precision and *F*-score, to measure the goodness of the results of IR queries. The Lucene open source indexing and search engine and its extension called Solr was discussed. Relevance feedback was briefly addressed—it is important to modify and improve the retrieval of pertinent information for the user through his interaction and engagement in the search process.

We provided in Section 27.7 a somewhat detailed introduction to analysis of the Web as it relates to information retrieval. We divided this treatment into the analysis of content, structure, and usage of the Web. Web search was discussed, including an analysis of the Web link structure (Section 27.7.3), including an introduction to algorithms for ranking the results from a Web search such as PageRank and HITS. Finally, we briefly discussed current trends, including faceted search, social search, and conversational search. We also presented probabilistic modeling of topics of documents and a popular technique called latent Dirichlet allocation. We ended the chapter with a discussion of question answering systems (Section 27.7.5), which are becoming very popular and use tools like Siri from Apple and Cortana from Microsoft.

This chapter provided an introductory treatment of a vast field. The interested reader should refer to the end-of-chapter bibliography for specialized texts on information retrieval and search engines.

## Review Questions

- 27.1.** What is structured data and what is unstructured data? Give an example of each from your experience.
- 27.2.** Give a general definition of *information retrieval* (IR). What does information retrieval involve when we consider information on the Web?
- 27.3.** Discuss the types of data and the types of users in today's information retrieval systems.

- 27.4. What is meant by *navigational*, *informational*, and *transformational search*?
- 27.5. What are the two main modes of interaction with an IR system? Describe and provide examples.
- 27.6. Explain the main differences between the database and IR systems mentioned in Table 27.1.
- 27.7. Describe the main components of the IR system as shown in Figure 27.1.
- 27.8. What are digital libraries? What types of data are typically found in them?
- 27.9. Name some digital libraries that you have accessed. What do they contain and how far back does the data go?
- 27.10. Give a brief history of IR and mention the landmark developments in this field.
- 27.11. What is the Boolean model of IR? What are its limitations?
- 27.12. What is the vector space model of IR? How does a vector get constructed to represent a document?
- 27.13. Define the TF-IDF scheme of determining the weight of a keyword in a document. Why is it necessary to include IDF in the weight of a term?
- 27.14. What are probabilistic and semantic models of IR?
- 27.15. Define *recall* and *precision* in IR systems.
- 27.16. Give the definition of *precision* and *recall* in a ranked list of results at position  $i$ .
- 27.17. How is an  $F$ -score defined as a metric of information retrieval? In what way does it account for both precision and recall?
- 27.18. What are the different types of queries in an IR system? Describe each with an example.
- 27.19. What are the approaches to processing phrase and proximity queries?
- 27.20. Describe the detailed IR process shown in Figure 27.2.
- 27.21. What is stopword removal and stemming? Why are these processes necessary for better information retrieval?
- 27.22. What is a thesaurus? How is it beneficial to IR?
- 27.23. What is information extraction? What are the different types of information extraction from structured text?
- 27.24. What are vocabularies in IR systems? What role do they play in the indexing of documents?
- 27.25. Gather five documents that contain about three sentences each and each contain some related content. Construct an inverted index of all important stems (keywords) from these documents.

- 27.26. Describe the process of constructing the result of a search request using an inverted index.
- 27.27. Define *relevance feedback*.
- 27.28. Describe the three types of Web analyses discussed in this chapter.
- 27.29. List the important tasks mentioned that are involved in analyzing Web content. Describe each in a couple of sentences.
- 27.30. What are the three categories of agent-based Web content analyses mentioned in this chapter?
- 27.31. What is the database-based approach to analyzing Web content? What are Web query systems?
- 27.32. What algorithms are popular in ranking or determining the importance of Web pages? Which algorithm was proposed by the founders of Google?
- 27.33. What is the basic idea behind the PageRank algorithm?
- 27.34. What are hubs and authority pages? How does the HITS algorithm use these concepts?
- 27.35. What can you learn from Web usage analysis? What data does it generate?
- 27.36. What mining operations are commonly performed on Web usage data? Give an example of each.
- 27.37. What are the applications of Web usage mining?
- 27.38. What is search relevance? How is it determined?
- 27.39. Define *faceted search*. Make up a set of facets for a database containing all types of buildings. For example, two facets could be “building value or price” and “building type (residential, office, warehouse, factory, and so on)”.
- 27.40. What is social search? What does collaborative social search involve?
- 27.41. Define and explain *conversational search*.
- 27.42. Define *topic modeling*.
- 27.43. How do question answering systems work?

## Selected Bibliography

Information retrieval and search technologies are active areas of research and development in industry and academia. There are many IR textbooks that provide detailed discussion of the materials that we have briefly introduced in this chapter. The book entitled *Search Engines: Information Retrieval in Practice* by Croft, Metzler, and Strohman (2009) gives a practical overview of search engine concepts and principles. *Introduction to Information Retrieval* by Manning, Raghavan, and Schütze (2008) is an authoritative book on information retrieval. Another introductory

textbook in IR is *Modern Information Retrieval* by Ricardo Baeza-Yates and Berthier Ribeiro-Neto (1999), which provides detailed coverage of various aspects of IR technology. Gerald Salton's (1968) and van Rijsbergen's (1979) classic books on information retrieval provide excellent descriptions of the foundational research done in the IR field until the late 1960s. Salton also introduced the vector space model as a model of IR. Manning and Schutze (1999) provide a good summary of natural language technologies and text preprocessing. "Interactive Information Retrieval in Digital Environments" by Xie (2008) provides a good human-centered approach to information retrieval. The book *Managing Gigabytes* by Witten, Moffat, and Bell (1999) provides detailed discussions for indexing techniques. The TREC book by Voorhees and Harman (2005) provides a description of test collection and evaluation procedures in the context of TREC competitions.

Broder (2002) classifies Web queries into three distinct classes—navigational, informational, and transactional—and presents a detailed taxonomy of Web search. Covi and Kling (1996) give a broad definition of digital libraries and discuss organizational dimensions of effective digital library use. Luhn (1957) did seminal work in IR at IBM in the 1950s on autoindexing and business intelligence. The SMART system (Salton et al. (1993)), developed at Cornell, was one of the earliest advanced IR systems that used fully automatic term indexing, hierarchical clustering, and document ranking by degree of similarity to the query. The SMART system represented documents and queries as weighted term vectors according to the vector space model.

Porter (1980) is credited with the weak and strong stemming algorithms that have become standards. Robertson (1997) developed a sophisticated weighting scheme in the City University of London Okapi system that became very popular in TREC competitions. Lenat (1995) started the Cyc project in the 1980s for incorporating formal logic and knowledge bases in information processing systems. Efforts toward creating the WordNet thesaurus continued in the 1990s and are still ongoing. WordNet concepts and principles are described in the book by Fellbaum (1998). Rocchio (1971) describes the relevance feedback algorithm, which is described in Salton's (1971) book on *The SMART Retrieval System—Experiments in Automatic Document Processing*.

Abiteboul, Buneman, and Suci (1999) provide an extensive discussion of data on the Web in their book that emphasizes semistructured data. Atzeni and Mendelzon (2000) wrote an editorial in the VLDB journal on databases and the Web. Atzeni et al. (2002) propose models and transformations for Web-based data. Abiteboul et al. (1997) propose the Lord query language for managing semistructured data.

Chakrabarti (2002) is an excellent book on knowledge discovery from the Web. The book by Liu (2006) consists of several parts, each providing a comprehensive overview of the concepts involved with Web data analysis and its applications. Excellent survey articles on Web analysis include Kosala and Blockeel (2000) and Liu et al. (2004). Etzioni (1996) provides a good starting point for understanding Web mining and describes the tasks and issues related to data mining on the World Wide Web. An excellent overview of the research issues, techniques, and development

efforts associated with Web content and usage analysis is presented by Cooley et al. (1997). Cooley (2003) focuses on mining Web usage patterns through the use of Web structure. Spiliopoulou (2000) describes Web usage analysis in detail. Web mining based on page structure is described in Madria et al. (1999) and Chakraborti et al. (1999). Algorithms to compute the rank of a Web page are given by Page et al. (1999), who describe the famous PageRank algorithm, and Kleinberg (1998), who presents the HITS algorithm.

Harth, Hose, and Schenkel (2014) present techniques for querying and managing linked data on the Web and show the potential of these techniques for research and commercial applications. Question answering technology is described in some detail by Ferrucci et al. (2010), who developed the IBM Watson system. Bikel and Zitouni (2012) is a comprehensive guide for developing robust and accurate multilingual NLP (natural language processing) systems. Blei, Ng, and Jordan (2003) provide an overview on topic modeling and latent Dirichlet allocation. For an in-depth, hands-on guide to Lucene and Solr technologies, refer to the upcoming book by Moczar (2015).

## Data Mining Concepts

Over the last several decades, many organizations have generated a large amount of machine-readable data in the form of files and databases. Existing database technology can process this data and supports query languages like SQL. However, SQL is a structured language that assumes the user is aware of the database schema. SQL supports operations of relational algebra that allow a user to select rows and columns of data from tables or join related information from tables based on common fields. In the next chapter, we will see that *data warehousing technology* affords several types of functionality: that of consolidation, aggregation, and summarization of data. Data warehouses let us view the same information along multiple dimensions. In this chapter, we will focus our attention on another very popular area of interest known as data mining. As the term connotes, **data mining** refers to the mining or discovery of new information in terms of patterns or rules from vast amounts of data. To be practically useful, data mining must be carried out efficiently on large files and databases. Although some data mining features are being provided in RDBMSs, data mining is *not* well-integrated with database management systems. The business world is presently fascinated by the potential of data mining, and the field of data mining is popularly called **business intelligence** or **data analytics**.

We will briefly review the basic concepts and principles of the extensive field of data mining, which uses techniques from such areas as machine learning, statistics, neural networks, and genetic algorithms. We will highlight the nature of the information that is discovered, the types of problems faced when trying to mine databases, and the applications of data mining. We will also survey the state of the art of a large number of commercial data mining tools (see Section 28.7) and describe a number of research advances that are needed to make this area viable.



## 28.1 Overview of Data Mining Technology

In reports such as the popular Gartner Report,<sup>1</sup> data mining has been hailed as one of the top technologies for the near future. In this section, we relate data mining to the broader area called *knowledge discovery* and contrast the two by means of an illustrative example.

### 28.1.1 Data Mining versus Data Warehousing

The goal of a data warehouse (see Chapter 29) is to support decision making with data. Data mining can be used in conjunction with a data warehouse to help with certain types of decisions. Data mining can be applied to operational databases with individual transactions. To make data mining more efficient, the data warehouse should have an aggregated or summarized collection of data. Data mining helps in extracting meaningful new patterns that cannot necessarily be found by merely querying or processing data or meta-data in the data warehouse. Therefore, data mining applications should be strongly considered early, during the design of a data warehouse. Also, data mining tools should be designed to facilitate their use in conjunction with data warehouses. In fact, for very large databases running into terabytes and even petabytes of data, successful use of data mining applications will depend first on the construction of a data warehouse.

### 28.1.2 Data Mining as a Part of the Knowledge Discovery Process

**Knowledge discovery in databases**, frequently abbreviated as **KDD**, typically encompasses more than data mining. The knowledge discovery process comprises six phases:<sup>2</sup> data selection, data cleansing, enrichment, data transformation or encoding, data mining, and the reporting and display of the discovered information.

As an example, consider a transaction database maintained by a specialty consumer goods retailer. Suppose the client data includes a customer name, zip code, phone number, date of purchase, item code, price, quantity, and total amount. A variety of new knowledge can be discovered by KDD processing on this client database. During *data selection*, data about specific items or categories of items, or from stores in a specific region or area of the country, may be selected. The *data cleansing* process then may correct invalid zip codes or eliminate records with incorrect phone prefixes. *Enrichment* typically enhances the data with additional sources of information. For example, given the client names and phone numbers, the store may purchase other data about age, income, and credit rating and append them to each record. *Data transformation* and encoding may be done to reduce the amount of

---

<sup>1</sup>The Gartner Report is one example of the many technology survey publications that corporate managers rely on to discuss and select data mining technology.

<sup>2</sup>This discussion is largely based on Adriaans and Zantinge (1996).

data. For instance, item codes may be grouped in terms of product categories into audio, video, supplies, electronic gadgets, camera, accessories, and so on. Zip codes may be aggregated into geographic regions, incomes may be divided into ranges, and so on. In Figure 29.1, we will show a process called extraction, transformation, and load (ETL) as a precursor to the data warehouse creation. If data mining is based on an existing warehouse for this retail store chain, we would expect that the cleaning has already been applied. It is only after such preprocessing that *data mining* techniques are used to mine different rules and patterns.

The result of mining may be to discover the following types of *new* information:

- **Association rules**—for example, whenever a customer buys video equipment, he or she also buys another electronic gadget.
- **Sequential patterns**—for example, suppose a customer buys a camera, and within three months he or she buys photographic supplies, then within six months he is likely to buy an accessory item. This defines a sequential pattern of transactions. A customer who buys more than twice in lean periods may be likely to buy at least once during the December holiday shopping period.
- **Classification trees**—for example, customers may be classified by frequency of visits, types of financing used, amount of purchase, or affinity for types of items; some revealing statistics may be generated for such classes.

As this retail store example shows, data mining must be preceded by significant data preparation before it can yield useful information that can directly influence business decisions.

The results of data mining may be reported in a variety of formats, such as listings, graphic outputs, summary tables, and visualizations.

### 28.1.3 Goals of Data Mining and Knowledge Discovery

Data mining is typically carried out with some end goals or applications. Broadly speaking, these goals fall into the following classes: prediction, identification, classification, and optimization.

- **Prediction.** Data mining can show how certain attributes within the data will behave in the future. Examples of predictive data mining include the analysis of buying transactions to predict what consumers will buy under certain discounts, how much sales volume a store will generate in a given period, and whether deleting a product line will yield more profits. In such applications, business logic is used coupled with data mining. In a scientific context, certain seismic wave patterns may predict an earthquake with high probability.
- **Identification.** Data patterns can be used to identify the existence of an item, an event, or an activity. For example, intruders trying to break a system may be identified by the programs executed, files accessed, and CPU time per session. In biological applications, existence of a gene may be

identified by certain sequences of nucleotide symbols in the DNA sequence. The area known as *authentication* is a form of identification. It ascertains whether a user is indeed a specific user or one from an authorized class, and it involves comparing parameters or images or signals against a database.

- **Classification.** Data mining can partition the data so that different classes or categories can be identified based on combinations of parameters. For example, customers in a supermarket can be categorized into discount-seeking shoppers, shoppers in a rush, loyal regular shoppers, shoppers attached to name brands, and infrequent shoppers. This classification may be used in different analyses of customer buying transactions as a post-mining activity. Sometimes classification based on common domain knowledge is used as an input to decompose the mining problem and make it simpler. For instance, health foods, party foods, and school lunch foods are distinct categories in the supermarket business. It makes sense to analyze relationships within and across categories as separate problems. Such categorization may be used to encode the data appropriately before subjecting it to further data mining.
- **Optimization.** One eventual goal of data mining may be to optimize the use of limited resources such as time, space, money, or materials and to maximize output variables such as sales or profits under a given set of constraints. As such, this goal of data mining resembles the objective function used in operations research problems that deals with optimization under constraints.

The term *data mining* is popularly used in a broad sense. In some situations, it includes statistical analysis and constrained optimization as well as machine learning. There is no sharp line separating data mining from these disciplines. It is beyond our scope, therefore, to discuss in detail the entire range of applications that make up this vast body of work. For a detailed understanding of the topic, readers are referred to specialized books devoted to data mining.

#### 28.1.4 Types of Knowledge Discovered during Data Mining

The term *knowledge* is broadly interpreted as involving some degree of intelligence. There is a progression from raw data to information to knowledge as we go through additional processing. Knowledge is often classified as inductive versus deductive. **Deductive knowledge** deduces new information based on applying *prespecified* logical rules of deduction on the given data. Data mining addresses **inductive knowledge**, which discovers new rules and patterns from the supplied data. Knowledge can be represented in many forms: In an unstructured sense, it can be represented by rules or propositional logic. In a structured form, it may be represented in decision trees, semantic networks, neural networks, or hierarchies of classes or frames. It is common to describe the knowledge discovered during data mining as follows:

- **Association rules.** These rules correlate the presence of a set of items with another range of values for another set of variables. Examples: (1) When a female retail shopper buys a handbag, she is likely to buy shoes. (2) An X-ray image containing characteristics a and b is likely to also exhibit characteristic c.

- **Classification hierarchies.** The goal is to work from an existing set of events or transactions to create a hierarchy of classes. Examples: (1) A population may be divided into five ranges of credit worthiness based on a history of previous credit transactions. (2) A model may be developed for the factors that determine the desirability of a store location on a 1–10 scale. (3) Mutual funds may be classified based on performance data using characteristics such as growth, income, and stability.
- **Sequential patterns.** A sequence of actions or events is sought. Example: If a patient underwent cardiac bypass surgery for blocked arteries and an aneurysm and later developed high blood urea within a year of surgery, he or she is likely to suffer from kidney failure within the next 18 months. Detecting sequential patterns is equivalent to detecting associations among events with certain temporal relationships.
- **Patterns within time series.** Similarities can be detected within positions of a **time series** of data, which is a sequence of data taken at regular intervals, such as daily sales or daily closing stock prices. Examples: (1) Stocks of a utility company, ABC Power, and a financial company, XYZ Securities, showed the same pattern during 2014 in terms of closing stock prices. (2) Two products show the same selling pattern in summer but a different one in winter. (3) A pattern in solar magnetic wind may be used to predict changes in Earth’s atmospheric conditions.
- **Clustering.** A given population of events or items can be partitioned (segmented) into sets of “similar” elements. Examples: (1) An entire population of treatment data on a disease may be divided into groups based on the similarity of side effects produced. (2) The adult population in the United States may be categorized into five groups from *most likely to buy* to *least likely to buy* a new product. (3) The Web accesses made by a collection of users against a set of documents (say, in a digital library) may be analyzed in terms of the keywords of documents to reveal clusters or categories of users.

For most applications, the desired knowledge is a combination of the above types. We expand on each of the above knowledge types in the following sections.

## 28.2 Association Rules

### 28.2.1 Market-Basket Model, Support, and Confidence

One of the major technologies in data mining involves the discovery of association rules. The database is regarded as a collection of transactions, each involving a set of items. A common example is that of **market-basket data**. Here the market basket corresponds to the sets of items a consumer buys in a supermarket during one visit. Consider four such transactions in a random sample shown in Figure 28.1.

An **association rule** is of the form  $X \Rightarrow Y$ , where  $X = \{x_1, x_2, \dots, x_n\}$ , and  $Y = \{y_1, y_2, \dots, y_m\}$  are sets of items, with  $x_i$  and  $y_j$  being distinct items for all  $i$  and all  $j$ . This

association states that if a customer buys  $X$ , he or she is also likely to buy  $Y$ . In general, any association rule has the form LHS (left-hand side)  $\Rightarrow$  RHS (right-hand side), where LHS and RHS are sets of items. The set  $LHS \cup RHS$  is called an **itemset**, the set of items purchased by customers. For an association rule to be of interest to a data miner, the rule should satisfy some interest measure. Two common interest measures are support and confidence.

The **support** for a rule  $LHS \Rightarrow RHS$  is with respect to the itemset; it refers to how frequently a specific itemset occurs in the database. That is, the support is the percentage of transactions that contain all of the items in the itemset  $LHS \cup RHS$ . If the support is low, it implies that there is no overwhelming evidence that items in  $LHS \cup RHS$  occur together because the itemset occurs in only a small fraction of transactions. Another term for support is *prevalence* of the rule.

The **confidence** is with regard to the implication shown in the rule. The confidence of the rule  $LHS \Rightarrow RHS$  is computed as the  $\text{support}(LHS \cup RHS) / \text{support}(LHS)$ . We can think of it as the probability that the items in RHS will be purchased given that the items in LHS are purchased by a customer. Another term for confidence is *strength* of the rule.

As an example of support and confidence, consider the following two rules:  $\text{milk} \Rightarrow \text{juice}$  and  $\text{bread} \Rightarrow \text{juice}$ . Looking at our four sample transactions in Figure 28.1, we see that the support of  $\{\text{milk}, \text{juice}\}$  is 50% and the support of  $\{\text{bread}, \text{juice}\}$  is only 25%. The confidence of  $\text{milk} \Rightarrow \text{juice}$  is 66.7% (meaning that, of three transactions in which milk occurs, two contain juice) and the confidence of  $\text{bread} \Rightarrow \text{juice}$  is 50% (meaning that one of two transactions containing bread also contains juice).

As we can see, support and confidence do not necessarily go hand in hand. The goal of mining association rules, then, is to generate all possible rules that exceed some minimum user-specified support and confidence thresholds. The problem is thus decomposed into two subproblems:

1. Generate all itemsets that have a support that exceeds the threshold. These sets of items are called **large** (or **frequent**) **itemsets**. Note that large here means large support.
2. For each large itemset, all the rules that have a minimum confidence are generated as follows: For a large itemset  $X$  and  $Y \subset X$ , let  $Z = X - Y$ ; then if  $\text{support}(X) / \text{support}(Z) > \text{minimum confidence}$ , the rule  $Z \Rightarrow Y$  (that is,  $X - Y \Rightarrow Y$ ) is a valid rule.

Generating rules by using all large itemsets and their supports is relatively straightforward. However, discovering all large itemsets together with the value for their

**Figure 28.1**

Sample transactions in market-basket model.

Transaction_id	Time	Items_bought
101	6:35	milk, bread, cookies, juice
792	7:38	milk, juice
1130	8:05	milk, eggs
1735	8:40	bread, cookies, coffee

support is a major problem if the cardinality of the set of items is very high. A typical supermarket has thousands of items. The number of distinct itemsets is  $2^m$ , where  $m$  is the number of items, and counting support for all possible itemsets becomes very computation intensive. To reduce the combinatorial search space, algorithms for finding association rules utilize the following properties:

- A subset of a large itemset must also be large (that is, each subset of a large itemset exceeds the minimum required support).
- Conversely, a superset of a small itemset is also small (implying that it does not have enough support).

The first property is referred to as **downward closure**. The second property, called the **antimonotonicity** property, helps to reduce the search space of possible solutions. That is, once an itemset is found to be small (not a large itemset), then any extension to that itemset, formed by adding one or more items to the set, will also yield a small itemset.

### 28.2.2 Apriori Algorithm

The first algorithm to use the downward closure and antimonotonicity properties was the **apriori algorithm**, shown as Algorithm 28.1.

We illustrate Algorithm 28.1 using the transaction data in Figure 28.1 using a minimum support of 0.5. The candidate 1-itemsets are {milk, bread, juice, cookies, eggs, coffee} and their respective supports are 0.75, 0.5, 0.5, 0.5, 0.25, and 0.25. The first four items qualify for  $L_1$  since each support is greater than or equal to 0.5. In the first iteration of the repeat-loop, we extend the frequent 1-itemsets to create the candidate frequent 2-itemsets,  $C_2$ .  $C_2$  contains {milk, bread}, {milk, juice}, {bread, juice}, {milk, cookies}, {bread, cookies}, and {juice, cookies}. Notice, for example, that {milk, eggs} does not appear in  $C_2$  since {eggs} is small (by the antimonotonicity property) and does not appear in  $L_1$ . The supports for the six sets contained in  $C_2$  are 0.25, 0.5, 0.25, 0.25, 0.5, and 0.25 and are computed by scanning the set of transactions. Only the second 2-itemset {milk, juice} and the fifth 2-itemset {bread, cookies} have support greater than or equal to 0.5. These two 2-itemsets form the frequent 2-itemsets,  $L_2$ .

#### Algorithm 28.1. Apriori Algorithm for Finding Frequent (Large) Itemsets

**Input:** Database of  $m$  transactions,  $D$ , and a minimum support,  $mins$ , represented as a fraction of  $m$ .

**Output:** Frequent itemsets,  $L_1, L_2, \dots, L_k$

**Begin** /\* steps or statements are numbered for better readability \*/

1. Compute  $\text{support}(i_j) = \text{count}(i_j)/m$  for each individual item,  $i_1, i_2, \dots, i_n$  by scanning the database once and counting the number of transactions that item  $i_j$  appears in (that is,  $\text{count}(i_j)$ );
2. The candidate frequent 1-itemset,  $C_1$ , will be the set of items  $i_1, i_2, \dots, i_n$ ;

3. The subset of items containing  $i_j$  from  $C_1$  where  $\text{support}(i_j) \geq \text{mins}$  becomes the frequent 1-itemset,  $L_1$ ;
  4.  $k = 1$ ;  
     termination = false;  
     **repeat**
    1.  $L_{k+1} = (\text{empty set})$ ;
    2. Create the candidate frequent  $(k+1)$ -itemset,  $C_{k+1}$ , by combining members of  $L_k$  that have  $k-1$  items in common (this forms candidate frequent  $(k+1)$ -itemsets by selectively extending frequent  $k$ -itemsets by one item);
    3. In addition, only consider as elements of  $C_{k+1}$  those  $k+1$  items such that every subset of size  $k$  appears in  $L_k$ ;
    4. Scan the database once and compute the support for each member of  $C_{k+1}$ ; if the support for a member of  $C_{k+1} \geq \text{mins}$  then add that member to  $L_{k+1}$ ;
    5. If  $L_{k+1}$  is empty then termination = true  
     else  $k = k + 1$ ;**until termination**;
- End**;

In the next iteration of the repeat-loop, we construct candidate frequent 3-itemsets by adding additional items to sets in  $L_2$ . However, for no extension of itemsets in  $L_2$  will all 2-item subsets be contained in  $L_2$ . For example, consider {milk, juice, bread}; the 2-itemset {milk, bread} is not in  $L_2$ , hence {milk, juice, bread} cannot be a frequent 3-itemset by the downward closure property. At this point the algorithm terminates with  $L_1$  equal to {{milk}, {bread}, {juice}, {cookies}} and  $L_2$  equal to {{milk, juice}, {bread, cookies}}.

Several other algorithms have been proposed to mine association rules. They vary mainly in terms of how the candidate itemsets are generated and how the supports for the candidate itemsets are counted. Some algorithms use data structures such as bitmaps and hashtrees to keep information about itemsets. Several algorithms have been proposed that use multiple scans of the database because the potential number of itemsets,  $2^m$ , can be too large to set up counters during a single scan. We will examine three improved algorithms (compared to the Apriori algorithm) for association rule mining: the sampling algorithm, the frequent-pattern tree algorithm, and the partition algorithm.

### 28.2.3 Sampling Algorithm

The main idea for the **sampling algorithm** is to select a small sample, one that fits in main memory, of the database of transactions and to determine the frequent itemsets from that sample. If those frequent itemsets form a superset of the frequent itemsets for the entire database, then we can determine the real frequent itemsets by scanning the remainder of the database in order to compute the exact support values for the superset itemsets. A superset of the frequent itemsets can usually be



found from the sample by using, for example, the apriori algorithm, with a lowered minimum support.

In rare cases, some frequent itemsets may be missed and a second scan of the database is needed. To decide whether any frequent itemsets have been missed, the concept of the *negative border* is used. The negative border with respect to a frequent itemset,  $S$ , and set of items,  $I$ , is the minimal itemsets contained in  $\text{PowerSet}(I)$  and not in  $S$ . The basic idea is that the negative border of a set of frequent itemsets contains the closest itemsets that could also be frequent. Consider the case where a set  $X$  is not contained in the frequent itemsets. If all subsets of  $X$  are contained in the set of frequent itemsets, then  $X$  would be in the negative border.

We illustrate this with the following example. Consider the set of items  $I = \{A, B, C, D, E\}$  and let the combined frequent itemsets of size 1 to 3 be  $S = \{\{A\}, \{B\}, \{C\}, \{D\}, \{AB\}, \{AC\}, \{BC\}, \{AD\}, \{CD\}, \{ABC\}\}$ . The negative border is  $\{\{E\}, \{BD\}, \{ACD\}\}$ . The set  $\{E\}$  is the only 1-itemset not contained in  $S$ ,  $\{BD\}$  is the only 2-itemset not in  $S$  but whose 1-itemset subsets are, and  $\{ACD\}$  is the only 3-itemset whose 2-itemset subsets are all in  $S$ . The negative border is important since it is necessary to determine the support for those itemsets in the negative border to ensure that no large itemsets are missed from analyzing the sample data.

Support for the negative border is determined when the remainder of the database is scanned. If we find that an itemset,  $X$ , in the negative border belongs in the set of all frequent itemsets, then there is a potential for a superset of  $X$  to also be frequent. If this happens, then a second pass over the database is needed to make sure that all frequent itemsets are found.

### 28.2.4 Frequent-Pattern (FP) Tree and FP-Growth Algorithm

The **frequent-pattern tree (FP-tree)** is motivated by the fact that apriori-based algorithms may generate and test a very large number of candidate itemsets. For example, with 1,000 frequent 1-itemsets, the apriori algorithm would have to generate

$$\binom{1000}{2}$$

or 499,500 candidate 2-itemsets. The **FP-growth algorithm** is one approach that eliminates the generation of a large number of candidate itemsets.

The algorithm first produces a compressed version of the database in terms of an FP-tree (frequent-pattern tree). The FP-tree stores relevant itemset information and allows for the efficient discovery of frequent itemsets. The actual mining process adopts a divide-and-conquer strategy, where the mining process is decomposed into a set of smaller tasks that each operates on a conditional FP-tree, a subset (projection) of the original tree. To start with, we examine how the FP-tree is constructed. The database is first scanned and the frequent 1-itemsets along with their support are computed. With this algorithm, the support is the *count* of transactions



containing the item rather than the fraction of transactions containing the item. The frequent 1-itemsets are then sorted in nonincreasing order of their support. Next, the root of the FP-tree is created with a NULL label. The database is scanned a second time and for each transaction  $T$  in the database, the frequent 1-itemsets in  $T$  are placed in order as was done with the frequent 1-itemsets. We can designate this sorted list for  $T$  as consisting of a first item, the head, and the remaining items, the tail. The itemset information (head, tail) is inserted into the FP-tree recursively, starting at the root node, as follows:

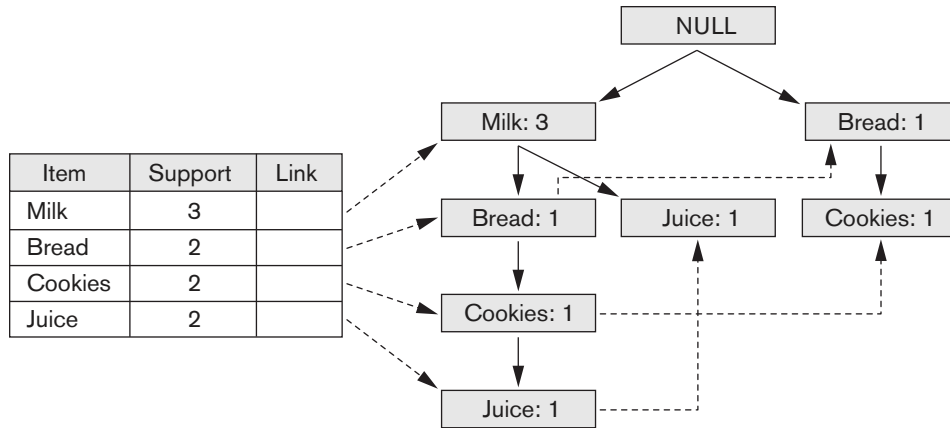
1. If the current node,  $N$ , of the FP-tree has a child with an item name = head, then increment the count associated with node  $N$  by 1, else create a new node,  $N$ , with a count of 1, link  $N$  to its parent and link  $N$  with the item header table (used for efficient tree traversal).
2. If the tail is nonempty, then repeat step (1) using as the sorted list only the tail, that is, the old head is removed and the new head is the first item from the tail and the remaining items become the new tail.

The item header table, created during the process of building the FP-tree, contains three fields per entry for each frequent item: item identifier, support count, and node link. The item identifier and support count are self-explanatory. The node link is a pointer to an occurrence of that item in the FP-tree. Since multiple occurrences of a single item may appear in the FP-tree, these items are linked together as a list where the start of the list is pointed to by the node link in the item header table. We illustrate the building of the FP-tree using the transaction data in Figure 28.1. Let us use a minimum support of 2. One pass over the four transactions yields the following frequent 1-itemsets with associated support:  $\{(milk, 3)\}$ ,  $\{(bread, 2)\}$ ,  $\{(cookies, 2)\}$ ,  $\{(juice, 2)\}$ . The database is scanned a second time and each transaction will be processed again.

For the first transaction, we create the sorted list,  $T = \{milk, bread, cookies, juice\}$ . The items in  $T$  are the frequent 1-itemsets from the first transaction. The items are ordered based on the nonincreasing ordering of the count of the 1-itemsets found in pass 1 (that is, milk first, bread second, and so on). We create a NULL root node for the FP-tree and insert *milk* as a child of the root, *bread* as a child of *milk*, *cookies* as a child of *bread*, and *juice* as a child of *cookies*. We adjust the entries for the frequent items in the item header table.

For the second transaction, we have the sorted list  $\{milk, juice\}$ . Starting at the root, we see that a child node with label *milk* exists, so we move to that node and update its count (to account for the second transaction that contains milk). We see that there is no child of the current node with label *juice*, so we create a new node with label *juice*. The item header table is adjusted.

The third transaction only has 1-frequent item,  $\{milk\}$ . Again, starting at the root, we see that the node with label *milk* exists, so we move to that node, increment its count, and adjust the item header table. The final transaction contains frequent items,  $\{bread, cookies\}$ . At the root node, we see that a child with label *bread* does not exist. Thus, we create a new child of the root, initialize its counter, and then



**Figure 28.2**  
FP-tree and item  
header table.

insert *cookies* as a child of this node and initialize its count. After the item header table is updated, we end up with the FP-tree and item header table as shown in Figure 28.2. If we examine this FP-tree, we see that it indeed represents the original transactions in a compressed format (that is, only showing the items from each transaction that are large 1-itemsets).

Algorithm 28.2 is used for mining the FP-tree for frequent patterns. With the FP-tree, it is possible to find all frequent patterns that contain a given frequent item by starting from the item header table for that item and traversing the node links in the FP-tree. The algorithm starts with a frequent 1-itemset (suffix pattern) and constructs its conditional pattern base and then its conditional FP-tree. The conditional pattern base is made up of a set of prefix paths, that is, where the frequent item is a suffix. For example, if we consider the item juice, we see from Figure 28.2 that there are two paths in the FP-tree that end with juice: (milk, bread, cookies, juice) and (milk, juice). The two associated prefix paths are (milk, bread, cookies) and (milk). The conditional FP-tree is constructed from the patterns in the conditional pattern base. The mining is recursively performed on this FP-tree. The frequent patterns are formed by concatenating the suffix pattern with the frequent patterns produced from a conditional FP-tree.

**Algorithm 28.2.** FP-Growth Algorithm for Finding Frequent Itemsets

**Input:** FP-tree and a minimum support, mins

**Output:** frequent patterns (itemsets)  
procedure FP-growth (tree, alpha);

**Begin**

if tree contains a single path  $P$  then  
for each combination, beta, of the nodes in the path  
generate pattern ( $\text{beta} \cup \text{alpha}$ )  
with support = minimum support of nodes in beta

```

else
    for each item,  $i$ , in the header of the tree do
        begin
            generate pattern  $\beta = (i \cup \alpha)$  with support =  $i$ .support;
            construct  $\beta$ 's conditional pattern base;
            construct  $\beta$ 's conditional FP-tree,  $\beta\_tree$ ;
            if  $\beta\_tree$  is not empty then
                FP-growth( $\beta\_tree$ ,  $\beta$ );
        end;
    End;

```

We illustrate the algorithm using the data in Figure 28.1 and the tree in Figure 28.2. The procedure FP-growth is called with the two parameters: the original FP-tree and NULL for the variable  $\alpha$ . Since the original FP-tree has more than a single path, we execute the else part of the first if statement. We start with the frequent item, juice. We will examine the frequent items in order of lowest support (that is, from the last entry in the table to the first). The variable  $\beta$  is set to juice with support equal to 2.

Following the node link in the item header table, we construct the conditional pattern base consisting of two paths (with juice as suffix). These are (milk, bread, cookies: 1) and (milk: 1). The conditional FP-tree consists of only a single node, milk: 2. This is due to a support of only 1 for node bread and cookies, which is below the minimal support of 2. The algorithm is called recursively with an FP-tree of only a single node (that is, milk: 2) and a  $\beta$  value of juice. Since this FP-tree only has one path, all combinations of  $\beta$  and nodes in the path are generated—that is, {milk, juice}—with support of 2.

Next, the frequent item, cookies, is used. The variable  $\beta$  is set to cookies with support = 2. Following the node link in the item header table, we construct the conditional pattern base consisting of two paths. These are (milk, bread: 1) and (bread: 1). The conditional FP-tree is only a single node, bread: 2. The algorithm is called recursively with an FP-tree of only a single node (that is, bread: 2) and a  $\beta$  value of cookies. Since this FP-tree only has one path, all combinations of  $\beta$  and nodes in the path are generated, that is, {bread, cookies} with support of 2. The frequent item, bread, is considered next. The variable  $\beta$  is set to bread with support = 2. Following the node link in the item header table, we construct the conditional pattern base consisting of one path, which is (milk: 1). The conditional FP-tree is empty since the count is less than the minimum support. Since the conditional FP-tree is empty, no frequent patterns will be generated.

The last frequent item to consider is milk. This is the top item in the item header table and as such has an empty conditional pattern base and empty conditional FP-tree. As a result, no frequent patterns are added. The result of executing the algorithm is the following frequent patterns (or itemsets) with their support: {{milk: 3}, {bread: 2}, {cookies: 2}, {juice: 2}, {milk, juice: 2}, {bread, cookies: 2}}.

### 28.2.5 Partition Algorithm

Another algorithm, called the **partition algorithm**,<sup>3</sup> is summarized below. If we are given a database with a small number of potential large itemsets, say, a few thousand, then the support for all of them can be tested in one scan by using a partitioning technique. Partitioning divides the database into nonoverlapping subsets; these are individually considered as separate databases and all large itemsets for that partition, called *local frequent itemsets*, are generated in one pass. The apriori algorithm can then be used efficiently on each partition if it fits entirely in main memory. Partitions are chosen in such a way that each partition can be accommodated in main memory. As such, a partition is read only once in each pass. The only caveat with the partition method is that the minimum support used for each partition has a slightly different meaning from the original value. The minimum support is based on the size of the partition rather than the size of the database for determining local frequent (large) itemsets. The actual support threshold value is the same as given earlier, but the support is computed only for a partition.

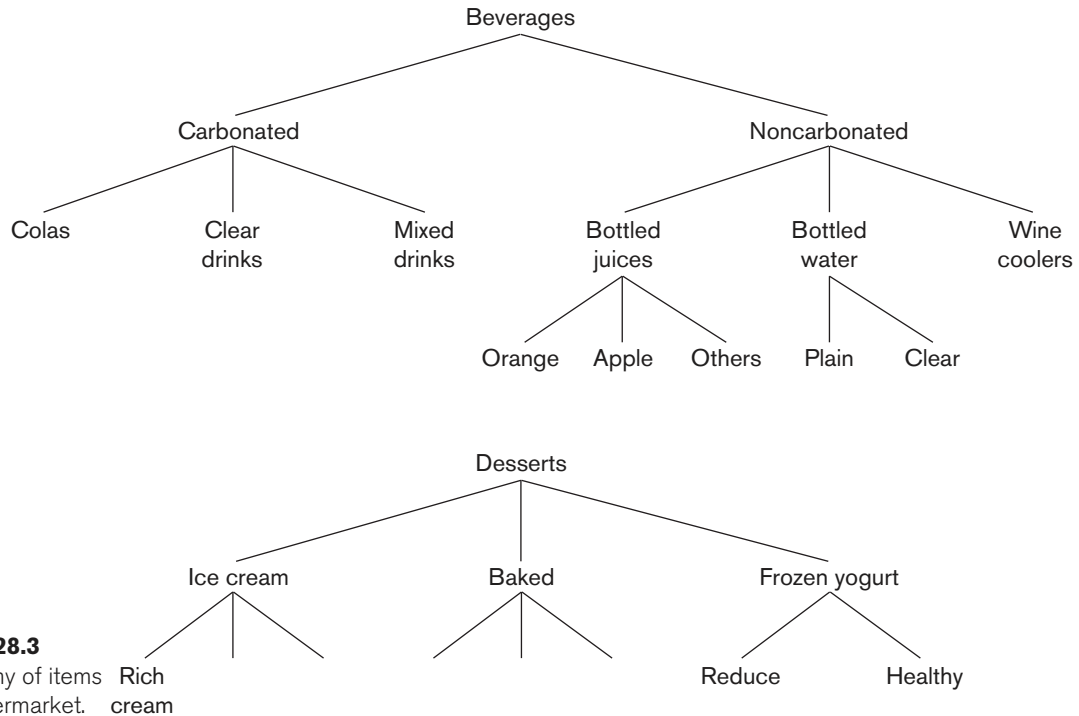
At the end of pass one, we take the union of all frequent itemsets from each partition. This forms the global candidate frequent itemsets for the entire database. When these lists are merged, they may contain some false positives. That is, some of the itemsets that are frequent (large) in one partition may not qualify in several other partitions and hence may not exceed the minimum support when the original database is considered. Note that there are no false negatives; no large itemsets will be missed. The global candidate large itemsets identified in pass one are verified in pass two; that is, their actual support is measured for the *entire* database. At the end of phase two, all global large itemsets are identified. The partition algorithm lends itself naturally to a parallel or distributed implementation for better efficiency. Further improvements to this algorithm have been suggested.<sup>4</sup>

### 28.2.6 Other Types of Association Rules

**Association Rules among Hierarchies.** There are certain types of associations that are particularly interesting for a special reason. These associations occur among hierarchies of items. Typically, it is possible to divide items among disjoint hierarchies based on the nature of the domain. For example, foods in a supermarket, items in a department store, or articles in a sports shop can be categorized into classes and subclasses that give rise to hierarchies. Consider Figure 28.3, which shows the taxonomy of items in a supermarket. The figure shows two hierarchies—beverages and desserts, respectively. The entire groups may not produce associations of the form *beverages*  $\Rightarrow$  *desserts*, or *desserts*  $\Rightarrow$  *beverages*. However, associations of the type *Healthy-brand frozen yogurt*  $\Rightarrow$  *bottled water*, or *Rich*

<sup>3</sup>See Savasere et al. (1995) for details of the algorithm, the data structures used to implement it, and its performance comparisons.

<sup>4</sup>See Cheung et al. (1996) and Lin and Dunham (1998).

**Figure 28.3**

Taxonomy of items  
in a supermarket.

Rich  
cream

cream-brand ice cream  $\Rightarrow$  wine cooler may produce enough confidence and support to be valid association rules of interest.

Therefore, if the application area has a natural classification of the itemsets into hierarchies, discovering associations *within* the hierarchies is of no particular interest. The ones of specific interest are associations *across* hierarchies. They may occur among item groupings at different levels.

**Multidimensional Associations.** Discovering association rules involves searching for patterns in a file. In Figure 28.1, we have an example of a file of customer transactions with three dimensions: Transaction\_id, Time, and Items\_bought. However, our data mining tasks and algorithms introduced up to this point only involve one dimension: Items\_bought. The following rule is an example of including the label of the single dimension: Items\_bought(milk)  $\Rightarrow$  Items\_bought(juice). It may be of interest to find association rules that involve multiple dimensions, for example, Time(6:30 ... 8:00)  $\Rightarrow$  Items\_bought(milk). Rules like these are called *multidimensional association rules*. The dimensions represent attributes of records of a file or, in terms of relations, columns of rows of a relation, and can be categorical or quantitative. Categorical attributes have a finite set of values that display no ordering relationship. Quantitative attributes are numeric and their values display an ordering relationship, for example,  $<$ . Items\_bought is an example of a categorical attribute and Transaction\_id and Time are quantitative.

One approach to handling a quantitative attribute is to partition its values into non-overlapping intervals that are assigned labels. This can be done in a static manner based on domain-specific knowledge. For example, a concept hierarchy may group values for Salary into three distinct classes: low income ( $0 < \text{Salary} < 29,999$ ), middle income ( $30,000 < \text{Salary} < 74,999$ ), and high income ( $\text{Salary} > 75,000$ ). From here, the typical apriori-type algorithm or one of its variants can be used for the rule mining since the quantitative attributes now look like categorical attributes. Another approach to partitioning is to group attribute values based on data distribution (for example, equi-depth partitioning) and to assign integer values to each partition. The partitioning at this stage may be relatively fine, that is, a larger number of intervals. Then during the mining process, these partitions may combine with other adjacent partitions if their support is less than some predefined maximum value. An apriori-type algorithm can be used here as well for the data mining.

**Negative Associations.** The problem of discovering a negative association is harder than that of discovering a positive association. A negative association is of the following type: *60% of customers who buy potato chips do not buy bottled water*. (Here, the 60% refers to the confidence for the negative association rule.) In a database with 10,000 items, there are 210,000 possible combinations of items, a majority of which do not appear even once in the database. If the absence of a certain item combination is taken to mean a negative association, then we potentially have millions and millions of negative association rules with RHSs that are of no interest at all. The problem, then, is to find only *interesting* negative rules. In general, we are interested in cases in which two specific sets of items appear very rarely in the same transaction. This poses two problems.

1. For a total item inventory of 10,000 items, the probability of any two being bought together is  $(1/10,000) * (1/10,000) = 10^{-8}$ . If we find the actual support for these two occurring together to be zero, that does not represent a significant departure from expectation and hence is not an interesting (negative) association.
2. The other problem is more serious. We are looking for item combinations with very low support, and there are millions and millions with low or even zero support. For example, a data set of 10 million transactions has most of the 2.5 billion pairwise combinations of 10,000 items missing. This would generate billions of useless rules.

Therefore, to make negative association rules interesting, we must use prior knowledge about the itemsets. One approach is to use hierarchies. Suppose we use the hierarchies of soft drinks and chips shown in Figure 28.4.



**Figure 28.4**  
Simple hierarchy of  
soft drinks and chips.

A strong positive association has been shown between soft drinks and chips. If we find a large support for the fact that when customers buy Days chips they predominantly buy Topsy and *not* Joke and *not* Wakeup, that would be interesting because we would normally expect that if there is a strong association between Days and Topsy, there should also be such a strong association between Days and Joke or Days and Wakeup.<sup>5</sup>

In the frozen yogurt and bottled water groupings shown in Figure 28.3, suppose the Reduce versus Healthy-brand division is 80–20 and the Plain and Clear brands division is 60–40 among respective categories. This would give a joint probability of Reduce frozen yogurt being purchased with Plain bottled water as 48% among the transactions containing a frozen yogurt and bottled water. If this support, however, is found to be only 20%, it would indicate a significant negative association among Reduce yogurt and Plain bottled water; again, that would be interesting.

The problem of finding negative association is important in the above situations given the domain knowledge in the form of item generalization hierarchies (that is, the beverage given and desserts hierarchies shown in Figure 28.3), the existing positive associations (such as between the frozen yogurt and bottled water groups), and the distribution of items (such as the name brands within related groups). The scope of discovery of negative associations is limited in terms of knowing the item hierarchies and distributions. Exponential growth of negative associations remains a challenge.

### 28.2.7 Additional Considerations for Association Rules

The mining of association rules in real-life databases is complicated by the following factors:

- The cardinality of itemsets in most situations is extremely large, and the volume of transactions is very high as well. Some operational databases in retailing and communication industries collect tens of millions of transactions per day.
- Transactions show variability in such factors as geographic location and seasons, making sampling difficult.
- Item classifications exist along multiple dimensions. Hence, driving the discovery process with domain knowledge, particularly for negative rules, is extremely difficult.
- Quality of data is variable; significant problems exist with missing, erroneous, conflicting, as well as redundant data in many industries.

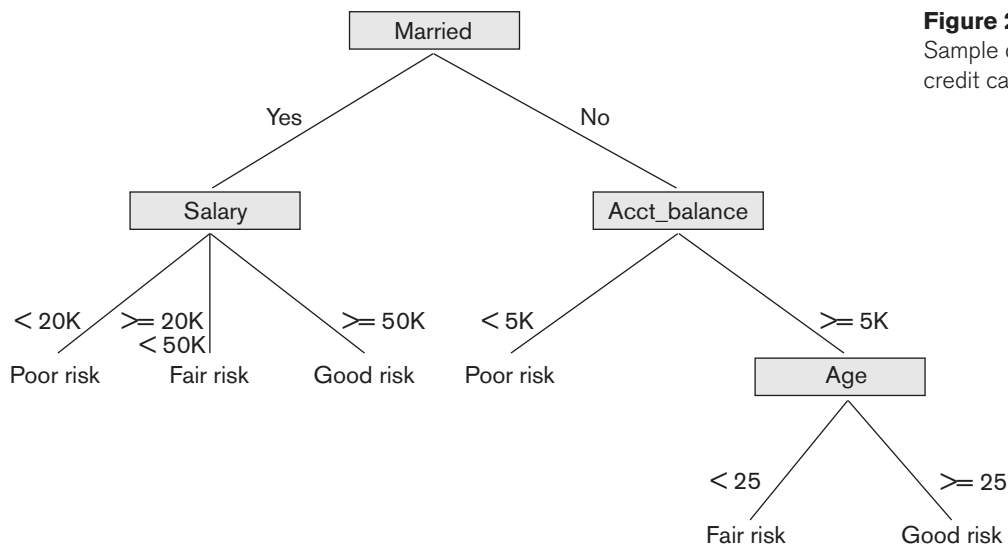
---

<sup>5</sup>For simplicity we are assuming a uniform distribution of transactions among members of a hierarchy.

## 28.3 Classification

**Classification** is the process of learning a model that describes different classes of data. The classes are predetermined. For example, in a banking application, customers who apply for a credit card may be classified as a *poor risk*, *fair risk*, or *good risk*. Hence this type of activity is also called **supervised learning**. Once the model is built, it can be used to classify new data. The first step—learning the model—is accomplished by using a training set of data that has already been classified. Each record in the training data contains an attribute, called the *class* label, which indicates which class the record belongs to. The model that is produced is usually in the form of a decision tree or a set of rules. Some of the important issues with regard to the model and the algorithm that produces the model include the model's ability to predict the correct class of new data, the computational cost associated with the algorithm, and the scalability of the algorithm.

We will examine the approach where our model is in the form of a decision tree. A **decision tree** is simply a graphical representation of the description of each class or, in other words, a representation of the classification rules. A sample decision tree is pictured in Figure 28.5. We see from Figure 28.5 that if a customer is *married* and if salary  $\geq 50K$ , then she is a good risk for a bank credit card. This is one of the rules that describe the class *good risk*. Traversing the decision tree from the root to each leaf node forms other rules for this class and the two other classes. Algorithm 28.3 shows the procedure for constructing a decision tree from a training data set. Initially, all training samples are at the root of the tree. The samples are partitioned recursively based on selected attributes. The attribute used at a node to partition the samples is the one with the best splitting criterion, for example, the one that maximizes the information gain measure.



**Figure 28.5**  
Sample decision tree for  
credit card applications.



**Algorithm 28.3.** Algorithm for Decision Tree Induction

**Input:** Set of training data records:  $R_1, R_2, \dots, R_m$  and set of attributes:  $A_1, A_2, \dots, A_n$

**Output:** Decision tree

procedure Build\_tree (records, attributes);

**Begin**

create a node  $N$ ;

if all records belong to the same class  $C$ , then

return  $N$  as a leaf node with class label  $C$ ;

if attributes is empty then

return  $N$  as a leaf node with class label  $C$ , such that the majority of records belong to it;

select attribute  $A_i$  (*with the highest information gain*) from attributes;

label node  $N$  with  $A_i$ ;

for each known value,  $v_j$ , of  $A_i$  do

**begin**

add a branch from node  $N$  for the condition  $A_i = v_j$ ;

$S_j$  = subset of records where  $A_i = v_j$ ;

if  $S_j$  is empty then

add a leaf,  $L$ , with class label  $C$ , such that the majority of records belong to it and return  $L$

else add the node returned by Build\_tree( $S_j$ , attributes –  $A_i$ );

**end;**

**End;**

Before we illustrate Algorithm 28.3, we will explain the **information gain** measure in more detail. The use of **entropy** as the information gain measure is motivated by the goal of minimizing the information needed to classify the sample data in the resulting partitions and thus minimizing the expected number of conditional tests needed to classify a new record. The expected information needed to classify training data of  $s$  samples, where the Class attribute has  $n$  values ( $v_1, \dots, v_n$ ) and  $s_i$  is the number of samples belonging to class label  $v_i$ , is given by

$$I(S_1, S_2, \dots, S_n) = - \sum_{i=1}^n p_i \log_2 p_i$$

where  $p_i$  is the probability that a random sample belongs to the class with label  $v_i$ . An estimate for  $p_i$  is  $s_i/s$ . Consider an attribute  $A$  with values  $\{v_1, \dots, v_m\}$  used as the test attribute for splitting in the decision tree. Attribute  $A$  partitions the samples into the subsets  $S_1, \dots, S_m$  where samples in each  $S_j$  have a value of  $v_j$  for attribute  $A$ . Each  $S_j$  may contain samples that belong to any of the classes. The number of samples in  $S_j$  that belong to class  $i$  can be denoted as  $s_{ij}$ . The entropy associated with using attribute  $A$  as the test attribute is defined as

$$E(A) = \sum_{j=1}^m \frac{S_{1j} + \dots + S_{nj}}{S} \times I(S_{1j}, \dots, S_{nj})$$

RID	Married	Salary	Acct_balance	Age	Loanworthy
1	no	$\geq 50K$	$< 5K$	$\geq 25$	yes
2	yes	$\geq 50K$	$\geq 5K$	$\geq 25$	yes
3	yes	20K. . . 50K	$< 5K$	$< 25$	no
4	no	$< 20K$	$\geq 5K$	$< 25$	no
5	no	$< 20K$	$< 5K$	$\geq 25$	no
6	yes	20K. . . 50K	$\geq 5K$	$\geq 25$	yes

**Figure 28.6**  
Sample training data  
for classification  
algorithm.

$I(s_{1j}, \dots, s_{nj})$  can be defined using the formulation for  $I(s_1, \dots, s_n)$  with  $p_i$  being replaced by  $p_{ij}$  where  $p_{ij} = s_{ij}/s_j$ . Now the information gain by partitioning on attribute  $A$ ,  $\text{Gain}(A)$ , is defined as  $I(s_1, \dots, s_n) - E(A)$ . We can use the sample training data from Figure 28.6 to illustrate the algorithm.

The attribute RID represents the record identifier used for identifying an individual record and is an internal attribute. We use it to identify a particular record in our example. First, we compute the expected information needed to classify the training data of 6 records as  $I(s_1, s_2)$  where there are two classes: the first class label value corresponds to *yes* and the second to *no*. So

$$I(3,3) = -0.5\log_2 0.5 - 0.5\log_2 0.5 = 1$$

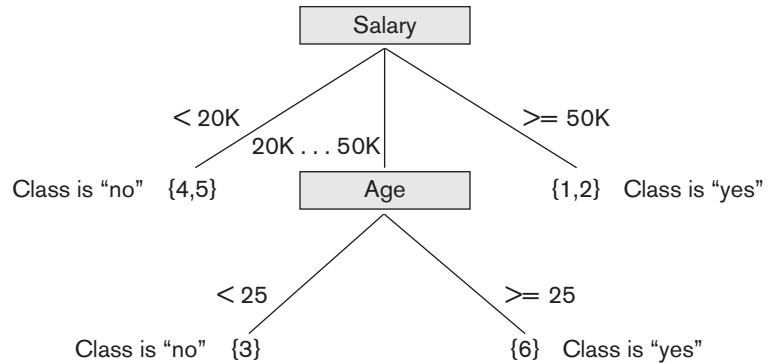
Now, we compute the entropy for each of the four attributes as shown below. For Married = yes, we have  $s_{11} = 2$ ,  $s_{21} = 1$  and  $I(s_{11}, s_{21}) = 0.92$ . For Married = no, we have  $s_{12} = 1$ ,  $s_{22} = 2$  and  $I(s_{12}, s_{22}) = 0.92$ . So the expected information needed to classify a sample using attribute Married as the partitioning attribute is

$$E(\text{Married}) = 3/6 I(s_{11}, s_{21}) + 3/6 I(s_{12}, s_{22}) = 0.92$$

The gain in information,  $\text{Gain}(\text{Married})$ , would be  $1 - 0.92 = 0.08$ . If we follow similar steps for computing the gain with respect to the other three attributes we end up with

$$\begin{array}{lll} E(\text{Salary}) = 0.33 & \text{and} & \text{Gain}(\text{Salary}) = 0.67 \\ E(\text{Acct\_balance}) = 0.92 & \text{and} & \text{Gain}(\text{Acct\_balance}) = 0.08 \\ E(\text{Age}) = 0.54 & \text{and} & \text{Gain}(\text{Age}) = 0.46 \end{array}$$

Since the greatest gain occurs for attribute Salary, it is chosen as the partitioning attribute. The root of the tree is created with label *Salary* and has three branches, one for each value of Salary. For two of the three values, that is,  $< 20K$  and  $> 50K$ , all the samples that are partitioned accordingly (records with RIDs 4 and 5 for  $< 20K$  and records with RIDs 1 and 2 for  $\geq 50K$ ) fall within the same class *loanworthy no* and *loanworthy yes*, respectively, for those two values. So we create a leaf node for each. The only branch that needs to be expanded is for the value 20K ... 50K with two samples, records with RIDs 3 and 6 in the training data. Continuing the process using these two records, we find that  $\text{Gain}(\text{Married})$  is 0,  $\text{Gain}(\text{Acct\_balance})$  is 1, and  $\text{Gain}(\text{Age})$  is 1.

**Figure 28.7**

Decision tree based on sample training data where the leaf nodes are represented by a set of RIDs of the partitioned records.

We can choose either Age or Acct\_balance since they both have the largest gain. Let us choose Age as the partitioning attribute. We add a node with label Age that has two branches, less than 25, and greater or equal to 25. Each branch partitions the remaining sample data such that one sample record belongs to each branch and hence one class. Two leaf nodes are created and we are finished. The final decision tree is pictured in Figure 28.7.

## 28.4 Clustering

The previous data mining task of classification deals with partitioning data based on using a preclassified training sample. However, it is often useful to partition data without having a training sample; this is also known as **unsupervised learning**. For example, in business, it may be important to determine groups of customers who have similar buying patterns, or in medicine, it may be important to determine groups of patients who show similar reactions to prescribed drugs. The goal of clustering is to place records into groups, such that records in a group are similar to each other and dissimilar to records in other groups. The groups are usually *disjoint*.

An important facet of clustering is the similarity function that is used. When the data is numeric, a similarity function based on distance is typically used. For example, the Euclidean distance can be used to measure similarity. Consider two  $n$ -dimensional data points (records)  $r_j$  and  $r_k$ . We can consider the value for the  $i$ th dimension as  $r_{ji}$  and  $r_{ki}$  for the two records. The Euclidean distance between points  $r_j$  and  $r_k$  in  $n$ -dimensional space is calculated as:

$$\text{Distance}(r_j, r_k) = \sqrt{|r_{j1} - r_{k1}|^2 + |r_{j2} - r_{k2}|^2 + \dots + |r_{jn} - r_{kn}|^2}$$

The smaller the distance between two points, the greater is the similarity as we think of them. A classic clustering algorithm is the  $k$ -means algorithm, Algorithm 28.4.

**Algorithm 28.4.** *k*-Means Clustering Algorithm**Input:** a database  $D$ , of  $m$  records,  $r_1, \dots, r_m$  and a desired number of clusters  $k$ **Output:** set of  $k$  clusters that minimizes the squared error criterion**Begin**

randomly choose  $k$  records as the centroids for the  $k$  clusters;  
 repeat  
   assign each record,  $r_i$ , to a cluster such that the distance between  $r_i$   
     and the cluster centroid (mean) is the smallest among the  $k$  clusters;  
   recalculate the centroid (mean) for each cluster based on the records  
     assigned to the cluster;  
 until no change;

**End;**

The algorithm begins by randomly choosing  $k$  records to represent the centroids (means),  $m_1, \dots, m_k$ , of the clusters,  $C_1, \dots, C_k$ . All the records are placed in a given cluster based on the distance between the record and the cluster mean. If the distance between  $m_i$  and record  $r_j$  is the smallest among all cluster means, then record  $r_j$  is placed in cluster  $C_i$ . Once all records have been initially placed in a cluster, the mean for each cluster is recomputed. Then the process repeats, by examining each record again and placing it in the cluster whose mean is closest. Several iterations may be needed, but the algorithm will converge, although it may terminate at a local optimum. The terminating condition is usually the squared-error criterion. For clusters  $C_1, \dots, C_k$  with means  $m_1, \dots, m_k$ , the error is defined as:

$$\text{Error} = \sum_{i=1}^k \sum_{\forall r_j \in C_i} \text{Distance}(r_j, m_i)^2$$

We will examine how Algorithm 28.4 works with the (two-dimensional) records in Figure 28.8. Assume that the number of desired clusters  $k$  is 2. Let the algorithm choose records with RID 3 for cluster  $C_1$  and RID 6 for cluster  $C_2$  as the initial cluster centroids. The remaining records will be assigned to one of those clusters during the first iteration of the repeat loop. The record with RID 1 has a distance from  $C_1$  of 22.4 and a distance from  $C_2$  of 32.0, so it joins cluster  $C_1$ . The record with RID 2 has

RID	Age	Years_of_service
1	30	5
2	50	25
3	50	15
4	25	5
5	30	10
6	55	25

**Figure 28.8**

Sample two-dimensional records for clustering example (the RID column is not considered).

a distance from  $C_1$  of 10.0 and a distance from  $C_2$  of 5.0, so it joins cluster  $C_2$ . The record with RID 4 has a distance from  $C_1$  of 25.5 and a distance from  $C_2$  of 36.6, so it joins cluster  $C_1$ . The record with RID 5 has a distance from  $C_1$  of 20.6 and a distance from  $C_2$  of 29.2, so it joins cluster  $C_1$ . Now, the new means (centroids) for the two clusters are computed. The mean for a cluster,  $C_i$ , with  $n$  records of  $m$  dimensions is the vector:

$$\bar{C}_i = \left( \frac{1}{n} \sum_{\forall r_j \in C_i} r_{ji}, \dots, \frac{1}{n} \sum_{\forall r_j \in C_i} r_{jm} \right)$$

The new mean for  $C_1$  is (33.75, 8.75) and the new mean for  $C_2$  is (52.5, 25). A second iteration proceeds and the six records are placed into the two clusters as follows: records with RIDs 1, 4, 5 are placed in  $C_1$  and records with RIDs 2, 3, 6 are placed in  $C_2$ . The mean for  $C_1$  and  $C_2$  is recomputed as (28.3, 6.7) and (51.7, 21.7), respectively. In the next iteration, all records stay in their previous clusters and the algorithm terminates.

Traditionally, clustering algorithms assume that the entire data set fits in main memory. More recently, researchers have developed algorithms that are efficient and are scalable for very large databases. One such algorithm is called BIRCH. BIRCH is a hybrid approach that uses both a hierarchical clustering approach, which builds a tree representation of the data, as well as additional clustering methods, which are applied to the leaf nodes of the tree. Two input parameters are used by the BIRCH algorithm. One specifies the amount of available main memory and the other is an initial threshold for the radius of any cluster. Main memory is used to store descriptive cluster information such as the center (mean) of a cluster and the radius of the cluster (clusters are assumed to be spherical in shape). The radius threshold affects the number of clusters that are produced. For example, if the radius threshold value is large, then few clusters of many records will be formed. The algorithm tries to maintain the number of clusters such that their radius is below the radius threshold. If available memory is insufficient, then the radius threshold is increased.

The BIRCH algorithm reads the data records sequentially and inserts them into an in-memory tree structure, which tries to preserve the clustering structure of the data. The records are inserted into the appropriate leaf nodes (potential clusters) based on the distance between the record and the cluster center. The leaf node where the insertion happens may have to split, depending upon the updated center and radius of the cluster and the radius threshold parameter. Additionally, when splitting, extra cluster information is stored, and if memory becomes insufficient, then the radius threshold will be increased. Increasing the radius threshold may actually produce a side effect of reducing the number of clusters since some nodes may be merged.

Overall, BIRCH is an efficient clustering method with a linear computational complexity in terms of the number of records to be clustered.

## 28.5 Approaches to Other Data Mining Problems

### 28.5.1 Discovery of Sequential Patterns

The discovery of sequential patterns is based on the concept of a sequence of itemsets. We assume that transactions such as the supermarket-basket transactions we discussed previously are ordered by time of purchase. That ordering yields a sequence of itemsets. For example, {milk, bread, juice}, {bread, eggs}, {cookies, milk, coffee} may be such a **sequence of itemsets** based on three visits by the same customer to the store. The **support** for a sequence  $S$  of itemsets is the percentage of the given set  $U$  of sequences of which  $S$  is a subsequence. In this example, {milk, bread, juice} {bread, eggs} and {bread, eggs} {cookies, milk, coffee} are considered **subsequences**. The problem of identifying sequential patterns, then, is to find all subsequences from the given sets of sequences that have a user-defined minimum support. The sequence  $S_1, S_2, S_3, \dots$  is a **predictor** of the fact that a customer who buys itemset  $S_1$  is likely to buy itemset  $S_2$  and then  $S_3$ , and so on. This prediction is based on the frequency (support) of this sequence in the past. Various algorithms have been investigated for sequence detection.

### 28.5.2 Discovery of Patterns in Time Series

**Time series** are sequences of events; each event may be a given fixed type of a transaction. For example, the closing price of a stock or a fund is an event that occurs every weekday for each stock and fund. The sequence of these values per stock or fund constitutes a time series. For a time series, one may look for a variety of patterns by analyzing sequences and subsequences as we did above. For example, we might find the period during which the stock rose or held steady for  $n$  days, or we might find the longest period over which the stock had a fluctuation of no more than 1% over the previous closing price, or we might find the quarter during which the stock had the most percentage gain or percentage loss. Time series may be compared by establishing measures of similarity to identify companies whose stocks behave in a similar fashion. Analysis and mining of time series is an extended functionality of temporal data management (see Chapter 26).

### 28.5.3 Regression

*Regression* is a special application of the classification rule. If a classification rule is regarded as a function over the variables that maps these variables into a target class variable, the rule is called a **regression rule**. A general application of regression occurs when, instead of mapping a tuple of data from a relation to a specific class, the value of a variable is predicted based on that tuple. For example, consider a relation

LAB\_TESTS (patient ID, test 1, test 2,  $\dots$ , test  $n$ )

which contains values that are results from a series of  $n$  tests for one patient. The target variable that we wish to predict is  $P$ , the probability of survival of the patient. Then the rule for regression takes the form:

$$\begin{aligned} &(\text{test 1 in range}_1) \text{ and } (\text{test 2 in range}_2) \text{ and } \dots (\text{test } n \text{ in range}_n) \Rightarrow P = x, \\ &\text{or } x < P \leq y \end{aligned}$$

The choice depends on whether we can predict a unique value of  $P$  or a range of values for  $P$ . If we regard  $P$  as a function:

$$P = f(\text{test 1, test 2, } \dots, \text{test } n)$$

the function is called a **regression function** to predict  $P$ . In general, if the function appears as

$$Y = f(X_1, X_2, \dots, X_n),$$

and  $f$  is linear in the domain variables  $x_i$ , the process of deriving  $f$  from a given set of tuples for  $\langle X_1, X_2, \dots, X_n, y \rangle$  is called **linear regression**. Linear regression is a commonly used statistical technique for fitting a set of observations or points in  $n$  dimensions with the target variable  $y$ .

Regression analysis is a very common tool for analysis of data in many research domains. The discovery of the function to predict the target variable is equivalent to a data mining operation.

### 28.5.4 Neural Networks

A **neural network** is a technique derived from artificial intelligence research that uses generalized regression and provides an iterative method to carry it out. Neural networks use the curve-fitting approach to infer a function from a set of samples. This technique provides a *learning approach*; it is driven by a test sample that is used for the initial inference and learning. With this kind of learning method, responses to new inputs may be able to be interpolated from the known samples. This interpolation, however, depends on the world model (internal representation of the problem domain) developed by the learning method.

Neural networks can be broadly classified into two categories: supervised and unsupervised networks. Adaptive methods that attempt to reduce the output error are **supervised learning** methods, whereas those that develop internal representations without sample outputs are called **unsupervised learning** methods.

Neural networks self-adapt; that is, they learn from information about a specific problem. They perform well on classification tasks and are therefore useful in data mining. Yet they are not without problems. Although they learn, they do not provide a good representation of *what* they have learned. Their outputs are highly quantitative and not easy to understand. As another limitation, the internal representations developed by neural networks are not unique. Also, in general, neural networks have trouble modeling time series data. Despite these shortcomings, they are popular and frequently used by several commercial vendors.

### 28.5.5 Genetic Algorithms

**Genetic algorithms** (GAs) are a class of randomized search procedures capable of adaptive and robust search over a wide range of search space topologies. Modeled after the adaptive emergence of biological species from evolutionary mechanisms, and introduced by Holland,<sup>6</sup> GAs have been successfully applied in such diverse fields as image analysis, scheduling, and engineering design.

Genetic algorithms extend the idea from human genetics of the four-letter alphabet (based on the A, C, T, G nucleotides) of the human DNA code. The construction of a genetic algorithm involves devising an alphabet that encodes the solutions to the decision problem in terms of strings of that alphabet. Strings are equivalent to individuals. A fitness function defines which solutions can survive and which cannot. The ways in which solutions can be combined are patterned after the cross-over operation of cutting and combining strings from a father and a mother. An initial population of a well-varied population is provided, and a game of evolution is played in which mutations occur among strings. They combine to produce a new generation of individuals; the fittest individuals survive and mutate until a family of successful solutions develops.

The solutions produced by GAs are distinguished from most other search techniques by the following characteristics:

- A GA search uses a set of solutions during each generation rather than a single solution.
- The search in the string-space represents a much larger parallel search in the space of encoded solutions.
- The memory of the search done is represented solely by the set of solutions available for a generation.
- A genetic algorithm is a randomized algorithm since search mechanisms use probabilistic operators.
- While progressing from one generation to the next, a GA finds near-optimal balance between knowledge acquisition and exploitation by manipulating encoded solutions.

Genetic algorithms are used for problem solving and clustering problems. Their ability to solve problems in parallel provides a powerful tool for data mining. The drawbacks of GAs include the large overproduction of individual solutions, the random character of the searching process, and the high demand on computer processing. In general, substantial computing power is required to achieve anything of significance with genetic algorithms.

---

<sup>6</sup>Holland's seminal work (1975) entitled *Adaptation in Natural and Artificial Systems* introduced the idea of genetic algorithms.



## 28.6 Applications of Data Mining

Data mining technologies can be applied to a large variety of decision-making contexts in business. In particular, areas of significant payoffs are expected to include the following:

- **Marketing.** Applications include analysis of consumer behavior based on buying patterns; determination of marketing strategies, including advertising, store location, and targeted mailing; segmentation of customers, stores, or products; and design of catalogs, store layouts, and advertising campaigns.
- **Finance.** Applications include analysis of creditworthiness of clients; segmentation of account receivables; performance analysis of finance investments like stocks, bonds, and mutual funds; evaluation of financing options; and fraud detection.
- **Manufacturing.** Applications involve optimization of resources like machines, personnel, and materials; and optimal design of manufacturing processes, shop-floor layouts, and product design, such as for automobiles based on customer requirements.
- **Healthcare.** Applications include discovery of patterns in radiological images, analysis of microarray (gene-chip) experimental data to cluster genes and to relate to symptoms or diseases, analysis of side effects of drugs and effectiveness of certain treatments, optimization of processes within a hospital, and analysis of the relationship between patient wellness data and doctor qualifications.

## 28.7 Commercial Data Mining Tools

Currently, commercial data mining tools use several common techniques to extract knowledge. These include association rules, clustering, neural networks, sequencing, and statistical analysis. We discussed these earlier. Also used are decision trees, which are a representation of the rules used in classification or clustering, and statistical analyses, which may include regression and many other techniques. Other commercial products use advanced techniques such as genetic algorithms, case-based reasoning, Bayesian networks, nonlinear regression, combinatorial optimization, pattern matching, and fuzzy logic. In this chapter, we have already discussed some of these.

Most data mining tools use the ODBC (Open Database Connectivity) interface. ODBC is an industry standard that works with databases; it enables access to data in most of the popular database programs such as Access, dBASE, Informix, Oracle, and SQL Server. Some of these software packages provide interfaces to specific database programs; the most common are Oracle, Access, and SQL Server. Most of the tools work in the Microsoft Windows environment and a few work in the UNIX operating system. The trend is for all products to operate under the Microsoft

Windows environment. One tool, Data Surveyor, mentions ODMG compliance; see Chapter 12, where we discussed the ODMG object-oriented standard.

In general, these programs perform sequential processing in a single machine. Many of these products work in the client/server mode. Some products incorporate parallel processing in parallel computer architectures and work as a part of online analytical processing (OLAP) tools.

### 28.7.1 User Interface

Most of the tools run in a graphical user interface (GUI) environment. Some products include sophisticated visualization techniques to view data and rules (for example, SGI's MineSet), and are even able to manipulate data this way interactively. Text interfaces are rare and are more common in tools available for UNIX, such as IBM's Intelligent Miner.

### 28.7.2 Application Programming Interface

Usually, the application programming interface (API) is an optional tool. Most products do not permit using their internal functions. However, some of them allow the application programmer to reuse their code. The most common interfaces are C libraries and dynamic link libraries (DLLs). Some tools include proprietary database command languages.

Table 28.1 lists 11 representative data mining tools. To date, there are hundreds of commercial data mining products available worldwide. Non-U.S. products include Data Surveyor from the Netherlands and PolyAnalyst from Russia.

### 28.7.3 Future Directions

Data mining tools are continually evolving, building on ideas from the latest scientific research. Many of these tools incorporate the latest algorithms taken from artificial intelligence (AI), statistics, and optimization. Currently, fast processing is done using modern database techniques—such as distributed processing—in client/server architectures, in parallel databases, and in data warehousing. For the future, the trend is toward developing Internet capabilities more fully. Additionally, hybrid approaches will become commonplace, and processing will be done using all resources available. Processing will take advantage of both parallel and distributed computing environments. This shift is especially important because modern databases contain very large amounts of information.

The primary direction for data mining is to analyze terabytes and petabytes of data in the so-called big data systems that we presented in Chapter 25. These systems are being equipped with their own tools and libraries for data mining, such as Mahout, which runs on top of Hadoop, which we described in detail. The data mining area will also be closely tied to data that will be housed in the cloud in data warehouses

**Table 28.1** Some Representative Data Mining Tools

Company	Product	Technique	Platform	Interface*
AcknoSoft	Kate	Decision trees, case-based reasoning	Windows UNIX	Microsoft Access
Angoss	Knowledge SEEKER	Decision trees, statistics	Windows	ODBC
Business Objects	Business Miner	Neural nets, machine learning	Windows	ODBC
CrossZ	QueryObject	Statistical analysis, optimization algorithm	Windows MVS UNIX	ODBC
Data Distilleries	Data Surveyor	Comprehensive; can mix different types of data mining	UNIX	ODBC and ODMG-compliant
DBMiner Technology Inc.	DBMiner	OLAP analysis, associations, classification, clustering algorithms	Windows	Microsoft 7.0 OLAP
IBM	Intelligent Miner	Classification, association rules, predictive models	UNIX (AIX)	IBM DB2
Megaputer Intelligence	PolyAnalyst	Symbolic knowledge acquisition, evolutionary programming	Windows OS/2	ODBC Oracle DB2
NCR	Management Discovery Tool (MDT)	Association rules	Windows	ODBC
Purple Insight	MineSet	Decision trees, association rules	UNIX (Irix)	Oracle Sybase Informix
SAS	Enterprise Miner	Decision trees, association rules, Nneural nets, regression, clustering	UNIX (Solaris) Windows Macintosh	ODBC Oracle AS/400

\*ODBC: Open Database Connectivity

ODMG: Object Data Management Group

and brought into service for mining operations as needed using OLAP (online analytical processing) servers. Not only are multimedia databases growing, but, in addition, image storage and retrieval are slow operations. Also, the cost of secondary storage is decreasing, so massive information storage will be feasible, even for small companies. Thus, data mining programs will have to deal with larger sets of data of more companies.

Most of data mining software will use the ODBC standard to extract data from business databases; proprietary input formats can be expected to disappear. There is a definite need to include nonstandard data, including images and other multimedia data, as source data for data mining.

## 28.8 Summary

In this chapter, we surveyed the important discipline of data mining, which uses database technology to discover additional knowledge or patterns in the data. We gave an illustrative example of knowledge discovery in databases, which has a wider scope than data mining. For data mining, among the various techniques, we focused on the details of association rule mining, classification, and clustering. We presented algorithms in each of these areas and illustrated with examples of how those algorithms work.

A variety of other techniques, including the AI-based neural networks and genetic algorithms, were also briefly discussed. Active research is ongoing in data mining, and we have outlined some of the expected research directions. In the future database technology products market, a great deal of data mining activity is expected. We summarized 11 out of several hundred data mining tools available; future research is expected to extend the number and functionality significantly.

## Review Questions

- 28.1. What are the different phases of the knowledge discovery from databases? Describe a complete application scenario in which new knowledge may be mined from an existing database of transactions.
- 28.2. What are the goals or tasks that data mining attempts to facilitate?
- 28.3. What are the five types of knowledge produced from data mining?
- 28.4. What are association rules as a type of knowledge? Define *support* and *confidence*, and use these definitions to define an association rule.
- 28.5. What is the downward closure property? How does it aid in developing an efficient algorithm for finding association rules; that is, with regard to finding large itemsets?
- 28.6. What was the motivating factor for the development of the FP-tree algorithm for association rule mining?
- 28.7. Describe an association rule among hierarchies and provide an example.
- 28.8. What is a negative association rule in the context of the hierarchy in Figure 28.3?
- 28.9. What are the difficulties of mining association rules from large databases?
- 28.10. What are classification rules, and how are decision trees related to them?
- 28.11. What is entropy, and how is it used in building decision trees?
- 28.12. How does clustering differ from classification?
- 28.13. Describe neural networks and genetic algorithms as techniques for data mining. What are the main difficulties in using these techniques?

## Exercises

**28.14.** Apply the apriori algorithm to the following data set.

Trans_id	Items_purchased
101	milk, bread, eggs
102	milk, juice
103	juice, butter
104	milk, bread, eggs
105	coffee, eggs
106	coffee
107	coffee, juice
108	milk, bread, cookies, eggs
109	cookies, butter
110	milk, bread

The set of items is {milk, bread, cookies, eggs, butter, coffee, juice}. Use 0.2 for the minimum support value.

- 28.15.** Show two rules that have a confidence of 0.7 or greater for an itemset containing three items from Exercise 28.14.
- 28.16.** For the partition algorithm, prove that any frequent itemset in the database must appear as a local frequent itemset in at least one partition.
- 28.17.** Show the FP-tree that would be made for the data from Exercise 28.14.
- 28.18.** Apply the FP-growth algorithm to the FP-tree from Exercise 28.17 and show the frequent itemsets.
- 28.19.** Apply the classification algorithm to the following set of data records. The class attribute is Repeat\_customer.

RID	Age	City	Gender	Education	Repeat_customer
101	20 ... 30	NY	F	college	YES
102	20 ... 30	SF	M	graduate	YES
103	31 ... 40	NY	F	college	YES
104	51 ... 60	NY	F	college	NO
105	31 ... 40	LA	M	high school	NO
106	41 ... 50	NY	F	college	YES
107	41 ... 50	NY	F	graduate	YES
108	20 ... 30	LA	M	college	YES
109	20 ... 30	NY	F	high school	NO
110	20 ... 30	NY	F	college	YES

**28.20.** Consider the following set of two-dimensional records:

RID	Dimension1	Dimension2
1	8	4
2	5	4
3	2	4
4	2	6
5	2	8
6	8	6

Also consider two different clustering schemes: (1) where Cluster<sub>1</sub> contains records {1, 2, 3} and Cluster<sub>2</sub> contains records {4, 5, 6}, and (2) where Cluster<sub>1</sub> contains records {1, 6} and Cluster<sub>2</sub> contains records {2, 3, 4, 5}. Which scheme is better and why?

**28.21.** Use the  $k$ -means algorithm to cluster the data from Exercise 28.20. We can use a value of 3 for  $K$ , and we can assume that the records with RIDs 1, 3, and 5 are used for the initial cluster centroids (means).

**28.22.** The  $k$ -means algorithm uses a similarity metric of distance between a record and a cluster centroid. If the attributes of the records are not quantitative but categorical in nature, such as `Income_level` with values {low, medium, high} or `Married` with values {Yes, No} or `State_of_residence` with values {Alabama, Alaska, ... , Wyoming}, then the distance metric is not meaningful. Define a more suitable similarity metric that can be used for clustering data records that contain categorical data.

## Selected Bibliography

Literature on data mining comes from several fields, including statistics, mathematical optimization, machine learning, and artificial intelligence. Chen et al. (1996) give a good summary of the database perspective on data mining. The book by Han and Kamber (2006) is an excellent text and describes in detail the different algorithms and techniques used in the data mining area. Work at IBM Almaden Research has produced a large number of early concepts and algorithms as well as results from some performance studies. Agrawal et al. (1993) report the first major study on association rules. Their apriori algorithm for market-basket data in Agrawal and Srikant (1994) is improved by using partitioning in Savasere et al. (1995); Toivonen (1996) proposes sampling as a way to reduce the processing effort. Cheung et al. (1996) extends the partitioning to distributed environments; Lin and Dunham (1998) propose techniques to overcome problems with data skew. Agrawal et al. (1993b) discuss the performance perspective on association rules. Mannila et al. (1994), Park et al. (1995), and Amir et al. (1997) present additional efficient algorithms related to association rules. Han et al. (2000) present the FP-tree algorithm

discussed in this chapter. Srikant and Agrawal (1995) proposes mining generalized rules. Savasere et al. (1998) present the first approach to mining negative associations. Agrawal et al. (1996) describe the Quest system at IBM. Sarawagi et al. (1998) describe an implementation where association rules are integrated with a relational database management system. Piatetsky-Shapiro and Frawley (1992) have contributed papers from a wide range of topics related to knowledge discovery. Zhang et al. (1996) present the BIRCH algorithm for clustering large databases. Information about decision tree learning and the classification algorithm presented in this chapter can be found in Mitchell (1997).

Adriaans and Zantinge (1996), Fayyad et al. (1997), and Weiss and Indurkha (1998) are books devoted to the different aspects of data mining and its use in prediction. The idea of genetic algorithms was proposed by Holland (1975); a good survey of genetic algorithms appears in Srinivas and Patnaik (1994). Neural networks have a vast literature; a comprehensive introduction is available in Lippman (1987).

Tan, Steinbach, and Kumar (2006) provides a comprehensive introduction to data mining and has a detailed set of references. Readers are also advised to consult proceedings of two prominent annual conferences in data mining: the Knowledge Discovery and Data Mining Conference (KDD), which has been running since 1995, and the SIAM International Conference on Data Mining (SDM), which has been running since 2001. Links to past conferences may be found at <http://dblp.uni-trier.de>.

## Overview of Data Warehousing and OLAP

Data warehouses are databases that store and maintain analytical data separately from transaction-oriented databases for the purpose of decision support. Regular transaction-oriented databases store data for a limited period of time before the data loses its immediate usefulness and it is archived. On the other hand, data warehouses tend to keep years' worth of data in order to enable analysis of historical data. They provide storage, functionality, and responsiveness to queries beyond the capabilities of transaction-oriented databases. Accompanying this ever-increasing power is a great demand to improve the data access performance of databases. In modern organizations, users of data are often completely removed from the data sources. Many people only need read-access to data, but still need fast access to a larger volume of data than can conveniently be downloaded to their desktops. Often such data comes from multiple databases. Because many of the analyses performed are recurrent and predictable, software vendors and systems support staff are designing systems to support these functions. Data warehouses are modeled and structured differently, they use different types of technologies for storage and retrieval, and they are used by different types of users than transaction-oriented databases. Presently there is a great need to provide decision makers from middle management upward with information at the correct level of detail to support decision making. *Data warehousing*, *online analytical processing* (OLAP), and *data mining* provide this functionality. We gave an introduction to data mining techniques in Chapter 28. In this chapter, we give a broad overview of data warehousing and OLAP technologies.



## 29.1 Introduction, Definitions, and Terminology

In Chapter 1, we defined a *database* as a collection of related data and a *database system* as a database and database software together. A data warehouse is also a collection of information as well as a supporting system. However, a clear distinction exists. Traditional databases are transactional (relational, object-oriented, network, or hierarchical). *Data warehouses* have the distinguishing characteristic that they are mainly intended for decision-support applications. They are optimized for data retrieval, not routine transaction processing.

Because data warehouses have been developed in numerous organizations to meet particular needs, there is no single, canonical definition of the term *data warehouse*. Professional magazine articles and books in the popular press have elaborated on the meaning in a variety of ways. Vendors have capitalized on the popularity of the term to help market a variety of related products, and consultants have provided a large variety of services, all under the data warehousing banner. However, data warehouses are distinct from traditional databases in their structure, functioning, performance, and purpose.

W. H. Inmon<sup>1</sup> characterized a **data warehouse** as *a subject-oriented, integrated, nonvolatile, time-variant collection of data in support of management's decisions*. Data warehouses provide access to data for complex analysis, knowledge discovery, and decision making through **ad hoc** and canned queries. **Canned queries** refer to *a-priori* defined queries with parameters that may recur with high frequency. They support high-performance demands on an organization's data and information. Several types of applications—OLAP, DSS, and data mining applications—are supported. We define each of these next.

**OLAP (online analytical processing)** is a term used to describe the analysis of complex data from the data warehouse. In the hands of skilled knowledge workers, OLAP tools enable quick and straightforward querying of the analytical data stored in data warehouses and **data marts** (analytical databases similar to data warehouses but with a defined narrow scope).

**DSS (decision-support systems)**, also known as **EIS (or MIS)—executive information systems (or management information systems)**, not to be confused with enterprise integration systems—support an organization's leading decision makers with higher-level (analytical) data for complex and important decisions. Data mining (which we discussed in Chapter 28) is used for *knowledge discovery*, the ad hoc process of searching data for unanticipated new knowledge (not unlike looking for pearls of wisdom in an ocean of data).

Traditional databases support **online transaction processing (OLTP)**, which includes insertions, updates, and deletions while also supporting information query requirements. Traditional relational databases are optimized to process queries that

---

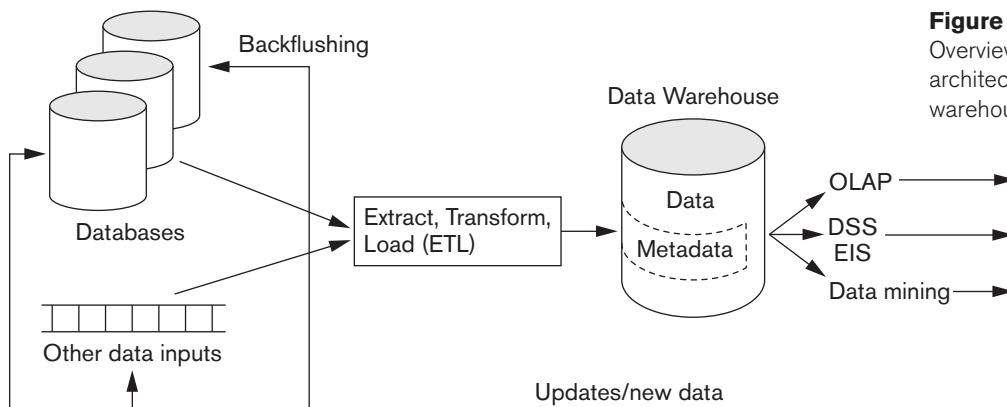
<sup>1</sup>Inmon (1992) is credited with initially using the term *warehouse*. Inmon et al. (2008) is titled "DW 2.0: The architecture for the next generation of Data Warehousing."

may touch a small part of the database and transactions that deal with insertions or updates of a few tuples per relation to process. Thus, they cannot be optimized for OLAP, DSS, or data mining. By contrast, data warehouses are designed precisely to support efficient extraction, processing, and presentation for analytic and decision-making purposes. In comparison to traditional databases, data warehouses generally contain very large amounts of data from multiple sources that may include databases from different data models and sometimes files acquired from independent systems and platforms.

## 29.2 Characteristics of Data Warehouses

To discuss data warehouses and distinguish them from transactional databases calls for an appropriate data model. The multidimensional data model (explained in more detail in Section 29.3) is a good fit for OLAP and decision-support technologies. In contrast to multidatabases, which provide access to disjoint and usually heterogeneous databases, a data warehouse is frequently a store of integrated data from multiple sources, processed for storage in a multidimensional model. Unlike most transactional databases, data warehouses typically support time series and trend analyses along with what-if or predictive-type analyses, all of which require more historical data than is generally maintained in transactional databases.

Compared with transactional databases, data warehouses are nonvolatile. This means that information in the data warehouse is typically not subject to modification and is often referred to as read/append/purge only. A data warehouse may be regarded as non-real-time with periodic insertions. In transactional systems, transactions are the unit and are the agent of change to the database; by contrast, data warehouse information is much more coarse-grained and is refreshed according to a careful choice of refresh policy, usually incremental. Warehouse insertions are handled by the warehouse's **ETL (extract, transform, load)** process, which does a large amount of preprocessing and which is shown in Figure 29.1. We can also describe



**Figure 29.1**  
Overview of the general architecture of a data warehouse.

data warehousing more generally as *a collection of decision-support technologies aimed at enabling the knowledge worker (executive, manager, analyst) to make better and faster decisions*.<sup>2</sup> Figure 29.1 gives an overview of the conceptual structure of a data warehouse. It shows the entire data warehousing process, which includes possible cleaning and reformatting of data before loading it into the warehouse. This process is handled by tools known as ETL (extraction, transformation, and loading) tools. At the back end of the process, OLAP, data mining, and DSS may generate new relevant information such as rules (or additional meta-data); this information is shown in Figure 29.1 as going back as additional data inputs into the warehouse. The figure also shows that data sources may include files.

The important characteristics of data warehouses that accompanied the definition of the term OLAP in 1993 included the following, and they are applicable even today:<sup>3</sup>

- Multidimensional conceptual view
- Unlimited dimensions and aggregation levels
- Unrestricted cross-dimensional operations
- Dynamic sparse matrix handling
- Client/server architecture
- Multiuser support
- Accessibility
- Transparency
- Intuitive data manipulation
- Inductive and deductive analysis
- Flexible distributed reporting

Because they encompass large volumes of data, data warehouses are generally an order of magnitude (sometimes two orders of magnitude) larger than the source databases. The sheer volume of data (likely to be in terabytes or even petabytes) is an issue that has been dealt with through enterprise-wide data warehouses, virtual data warehouses, logical data warehouses, and data marts:

- **Enterprise-wide data warehouses** are huge projects requiring massive investment of time and resources.
- **Virtual data warehouses** provide views of operational databases that are materialized for efficient access.
- **Logical data warehouses** use data federation, distribution, and virtualization techniques.
- **Data marts** generally are targeted to a subset of the organization, such as a department, and are more tightly focused.

---

<sup>2</sup>Chaudhuri and Dayal (1997) provide an excellent tutorial on the topic, with this as a starting definition.

<sup>3</sup>Codd and Salley (1993) coined the term *OLAP* and mentioned the characteristics listed here.

Other terms frequently encountered in the context of data warehousing are as follows:

- **Operational data store (ODS):** This term is commonly used for intermediate form of databases before they are cleansed, aggregated, and transformed into a data warehouse.
- **Analytical data store (ADS):** Those are the database that are built for the purpose of conducting data analysis. Typically, ODSs are reconfigured and repurposed into ADSs through the processes of cleansing, aggregation, and transformation.

### 29.3 Data Modeling for Data Warehouses

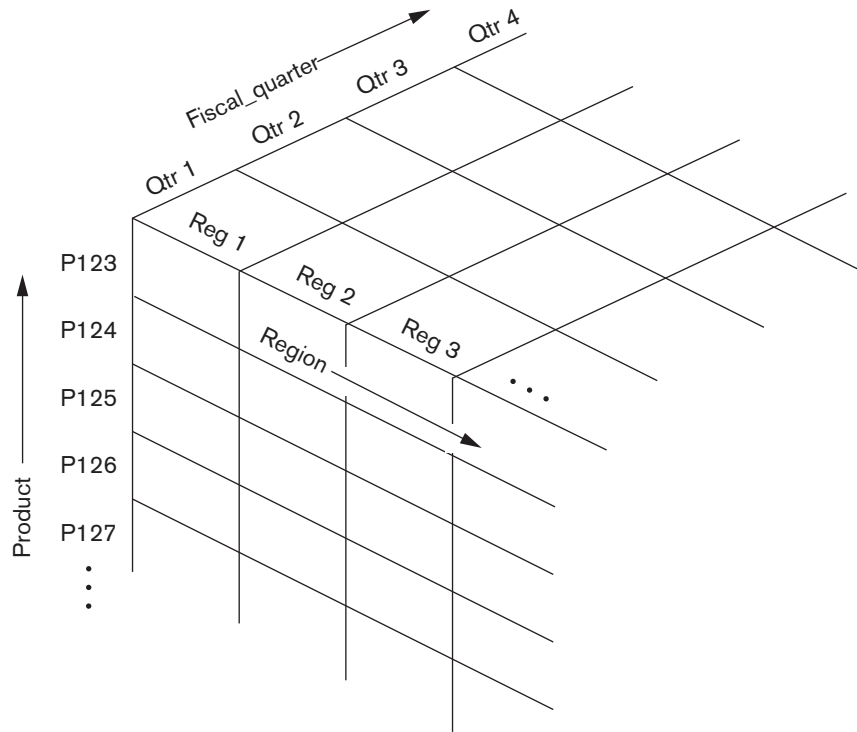
Multidimensional models take advantage of inherent relationships in data to populate data in multidimensional matrices called *data cubes*. (These may be called *hyper-cubes* if they have more than three dimensions.) For data that lends itself to multidimensional modeling, query performance in multidimensional matrices can be much better than in the relational data model. Three examples of dimensions in a corporate data warehouse are the corporation’s fiscal periods, products, and regions.

A standard spreadsheet is a two-dimensional matrix. One example would be a spreadsheet of regional sales by product for a particular time period. Products could be shown as rows, with columns comprising sales revenues for each region. (Figure 29.2 shows this two-dimensional organization.) Adding a time dimension, such as an organization’s fiscal quarters, would produce a three-dimensional matrix, which could be represented using a data cube.

Figure 29.3 shows a three-dimensional data cube that organizes product sales data by fiscal quarters and sales regions. Each cell could contain data for a specific product,

		Region			
		Reg 1	Reg 2	Reg 3	...
Product	P123				
	P124				
	P125				
	P126				
	...				

**Figure 29.2**  
A two-dimensional matrix model.

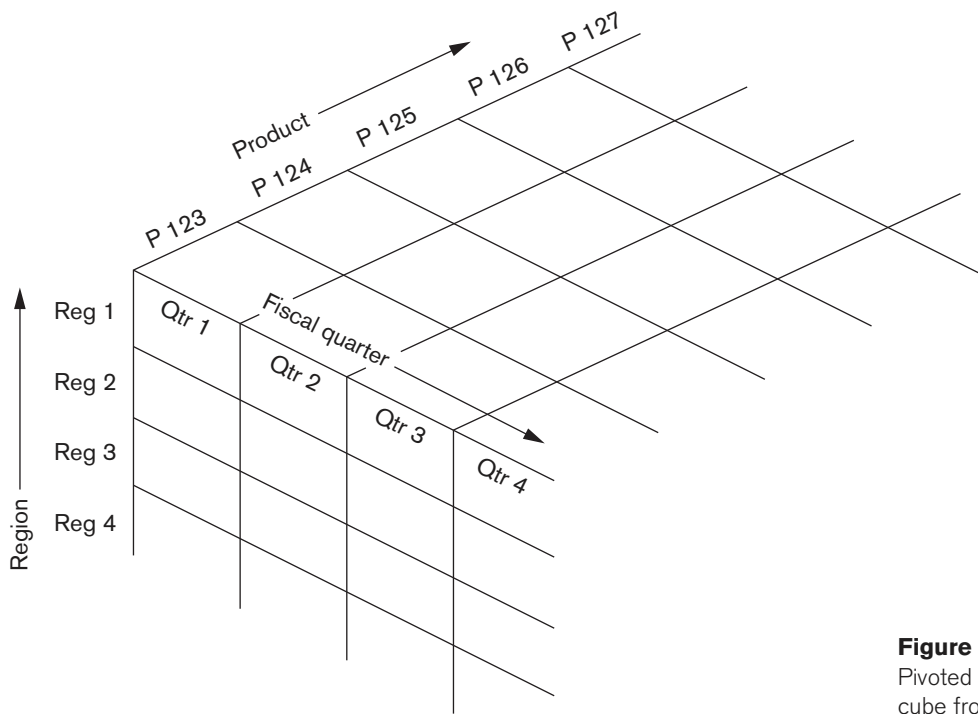
**Figure 29.3**

A three-dimensional data cube model.

specific fiscal quarter, and specific region. By including additional dimensions, a data hypercube could be produced, although more than three dimensions cannot be easily visualized or graphically presented. The data can be queried directly in any combination of dimensions, thus bypassing complex database queries. Tools exist for viewing data according to the user's choice of dimensions.

Changing from one-dimensional hierarchy (orientation) to another is easily accomplished in a data cube with a technique called **pivoting** (also called *rotation*). In this technique, the data cube can be thought of as rotating to show a different orientation of the axes. For example, you might pivot the data cube to show regional sales revenues as rows, the fiscal quarter revenue totals as columns, and the company's products in the third dimension (Figure 29.4). Hence, this technique is equivalent to having a regional sales table for each product separately, where each table shows quarterly sales for that product region by region. The term **slice** is used to refer to a two-dimensional view of a three- or higher-dimensional cube. The Product vs. Region 2-D view shown in Figure 29.2 is a slice of the 3-D cube shown in Figure 29.3. The popular term "slice and dice" implies a systematic reduction of a body of data into smaller chunks or views so that the information is made visible from multiple angles or viewpoints.

Multidimensional models lend themselves readily to hierarchical views in what is known as roll-up display and drill-down display. A **roll-up display** moves up the



**Figure 29.4**  
Pivoted version of the data cube from Figure 29.3.

hierarchy, grouping into larger units along a dimension (for example, summing weekly data by quarter or by year). Figure 29.5 shows a roll-up display that moves from individual products to a coarser grain of product categories. Shown in Figure 29.6, a **drill-down display** provides the opposite capability, furnishing a finer-grained view, perhaps disaggregating country sales by region and then

		Region →		
		Region 1	Region 2	Region 3
Product categories ↓	Products 1XX			
	Products 2XX			
	Products 3XX			
	Products 4XX			

**Figure 29.5**  
The roll-up operation.

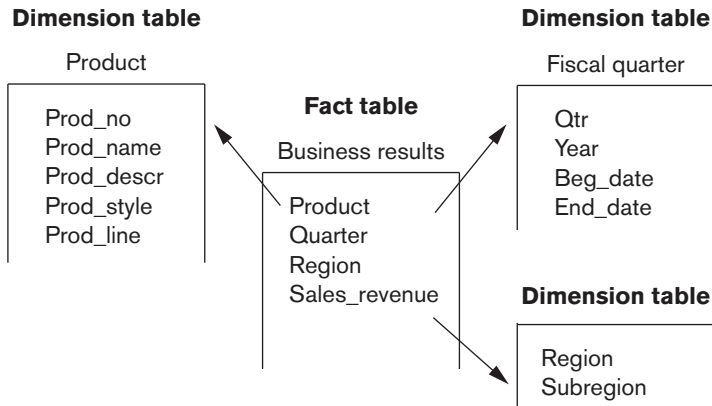
		Region 1				Region 2	
		Sub_reg 1	Sub_reg 2	Sub_reg 3	Sub_reg 4	Sub_reg 1	
P123 Styles	A						
	B						
	C						
	D						
P124 Styles	A						
	B						
	C						
P125 Styles	A						
	B						
	C						
	D						

**Figure 29.6**  
The drill-down operation.

regional sales by subregion and also breaking up products by styles. Typically, in a warehouse, the **drill-down** capability is limited to the lowest level of aggregated data stored in the warehouse. For example, compared to the data shown in Figure 29.6, lower- level data will correspond to something like “the total sales for style P123 substyle A color Black in zipcode 30022 of sub-region 1.” That level of aggregation may have been kept in the ODS. Some DBMSs like Oracle offer the “nested table” concept, which enables access to lower levels of data and thus makes the drill-down penetrate deeper.

The **multidimensional model** (also called the **dimensional model**)-involves two types of tables: dimension tables and fact tables. A **dimension table** consists of tuples of attributes of the dimension. A **fact table** can be thought of as having tuples, one per a recorded fact. This fact contains some measured or observed variable(s) and identifies it (them) with pointers to dimension tables. The fact table contains the data, and the dimensions identify each tuple in that data. Another way to look at a fact table is as an agglomerated view of the transaction data whereas each dimension table represents so-called “master data” that those transactions belonged to. In multidimensional database systems, the multidimensional model has been implemented as specialized software system known as a *multidimensional database*, which we do not discuss. Our treatment of the multidimensional model is based on storing the warehouse as a relational database in an RDBMS.

Figure 29.7 shows an example of a fact table that can be viewed from the perspective of multi-dimension tables. Two common multidimensional schemas are the star schema and the snowflake schema. The **star schema** consists of a fact table with a single table for each dimension (Figure 29.7). The **snowflake schema** is a variation on the star schema in which the dimensional tables from a star schema are organized

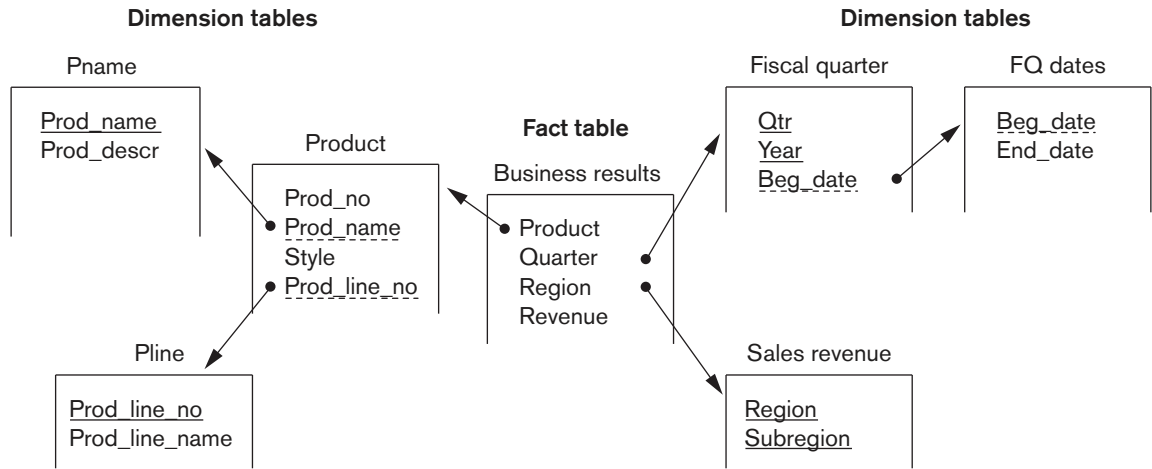


**Figure 29.7**  
A star schema with fact and dimensional tables.

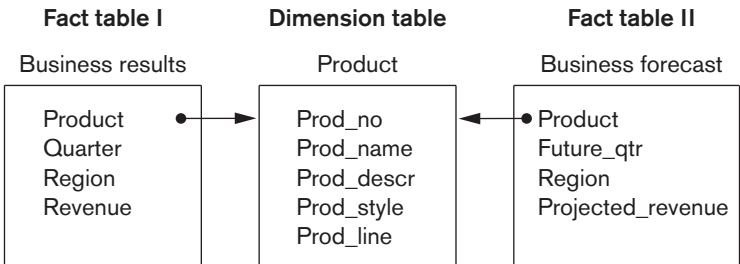
into a hierarchy by normalizing them (Figure 29.8). A **fact constellation** is a set of fact tables that share some dimension tables. Figure 29.9 shows a fact constellation with two fact tables, business results and business forecast. These share the dimension table called product. Fact constellations limit the possible queries for the warehouse.

Data warehouse storage also utilizes indexing techniques to support high-performance access (see Chapter 17 for a discussion of indexing). A technique called **bitmap indexing** constructs a bit vector for each value in a domain (column) being indexed. It works very well for domains of low cardinality. There is a 1 bit placed in

**Figure 29.8**  
A snowflake schema.







**Figure 29.9**  
A fact constellation.

the *j*th position in the vector if the *j*th row contains the value being indexed. For example, imagine an inventory of 100,000 cars with a bitmap index on car size. If there are four car sizes—economy, compact, mid-size, and full-size—there will be four bit vectors, each containing 100,000 bits (12.5kbytes) for a total index size of 50K. Bitmap indexing can provide considerable input/output and storage space advantages in low-cardinality domains. With bit vectors, a bitmap index can provide dramatic improvements in comparison, aggregation, and join performance. We showed an example of a query on a star schema in Section 19.8, and we also showed the star schema’s transformation for efficient execution that uses bitmap indexes.

In a star schema, dimensional data can be indexed to tuples in the fact table by **join indexing**. Join indexes are traditional indexes used to maintain relationships between primary key and foreign key values. They relate the values of a dimension of a star schema to rows in the fact table. Consider a sales fact table that has city and fiscal quarter as dimensions. If there is a join index on city, for each city the join index maintains the tuple IDs of tuples containing that city. Join indexes may involve multiple dimensions.

Data warehouse storage can facilitate access to summary data by taking further advantage of the nonvolatility of data warehouses and a degree of predictability of the analyses that will be performed using them. Two approaches have been used: (1) smaller tables that include summary data such as quarterly sales or revenue by product line, and (2) encoding of level (for example, weekly, quarterly, annual) into existing tables. The overhead of creating and maintaining such aggregations would likely be excessive in a dynamically changing, transaction-oriented database.

The purpose of **master data management (MDM)**, a popular concept within enterprises, is to define the standards, processes, policies, and governance related to the critical data entities of the organization. The dimension tables—which in a data warehouse physicalize concepts, such as customers, regions and product categories—represent essentially the master data. Since dimensions are shared across multiple facts or reporting data marts, data warehouse designers typically must spend a considerable amount of time cleansing and harmonizing these dimensions (i.e., reconciling definitional and notional differences across multiple source systems that the dimension data comes from). As such, table structures containing these dimensions become good candidates for special copies of master data that can be used in other environments.

## 29.4 Building a Data Warehouse

In constructing a data warehouse, builders should take a broad view of the anticipated use of the warehouse. There is no way to anticipate all possible queries or analyses during the design phase. However, the design should specifically support **ad hoc querying**; that is, accessing data with any combination of values for the attributes that would be meaningful in the dimension or fact tables. For example, a marketing-intensive consumer-products company would require different ways of organizing the data warehouse than would a nonprofit charity focused on fund raising. An appropriate schema should be chosen that reflects anticipated usage.

Acquisition of data for the warehouse involves the following steps:

1. The data must be extracted from multiple, heterogeneous sources; for example, databases or other data feeds such as those containing financial market data or environmental data.
2. Data must be formatted for consistency within the warehouse. Names, meanings, and domains of data from unrelated sources must be reconciled. For instance, subsidiary companies of a large corporation may have different fiscal calendars with quarters ending on different dates, making it difficult to aggregate financial data by quarter. Various credit cards may report their transactions differently, making it difficult to compute all credit sales. These format inconsistencies must be resolved.
3. The data must be cleaned to ensure validity. Data cleaning is an involved and complex process that has been identified as the largest labor-demanding component of data warehouse construction. For input data, cleaning must occur before the data is loaded into the warehouse. Since input data must be examined and formatted consistently, data warehouse builders should take this opportunity to check each input for validity and quality. Recognizing erroneous and incomplete data is difficult to automate, and cleaning that requires automatic error correction can be even tougher. Some aspects, such as domain checking, are easily coded into data cleaning routines, but automatic recognition of other data problems can be more challenging. (For example, one might require that City = 'San Francisco' together with State = 'CT' be recognized as an incorrect combination.) After such problems have been taken care of, similar data from different sources must be coordinated for loading into the warehouse. As data managers in the organization discover that their data is being cleaned for input into the warehouse, they will likely want to upgrade their data with the cleaned data. The process of returning cleaned data to the source is called **backflushing** (see Figure 29.1).
4. The data must be fitted into the data model of the warehouse. Data from the various sources must be represented in the data model of the warehouse. Data may have to be converted from relational, object-oriented, or legacy databases (network and/or hierarchical) to a multidimensional model.
5. The data must be loaded into the warehouse. The sheer volume of data in the warehouse makes loading the data a significant task. Monitoring tools

for loads as well as methods to recover from incomplete or incorrect loads are required. With the huge volume of data in the warehouse, incremental updating is usually the only feasible approach. The refresh policy will probably emerge as a compromise that takes into account the answers to the following questions:

- ❑ How up-to-date must the data be?
- ❑ Can the warehouse go offline, and for how long?
- ❑ What are the data interdependencies?
- ❑ What is the storage availability?
- ❑ What are the distribution requirements (such as for replication and partitioning)?
- ❑ What is the loading time (including cleaning, formatting, copying, transmitting, and overhead such as index rebuilding)?

Data in a warehouse can come from multiple sources, geographies, and/or time zones. Data loads, therefore, need to be carefully planned and staged. The order in which data is loaded into the warehouse is critical; failure to load data in the correct order could lead to integrity constraints or semantic rule violations, both of which could cause load failures. For example, master data (whether new or changed) such as Customer and Product must be loaded prior to the transactions that contain them; and invoice data must be loaded before the billing data that references it.

As we have said, databases must strike a balance between efficiency in transaction processing and support for query requirements (ad hoc user requests), but a data warehouse is typically optimized for access from a decision maker's needs. Data storage in a data warehouse reflects this specialization and involves the following processes:

- Storing the data according to the data model of the warehouse
- Creating and maintaining required data structures
- Creating and maintaining appropriate access paths
- Providing for time-variant data as new data are added
- Supporting the updating of warehouse data
- Refreshing the data
- Purging data

Although adequate time can be devoted initially to constructing the warehouse, the sheer volume of data in the warehouse generally makes it impossible to simply reload the warehouse in its entirety later on. Alternatives include selective (partial) refreshing of data and separate warehouse versions (which requires double storage capacity for the warehouse). When the warehouse uses an incremental data refreshing mechanism, data may need to be purged periodically; for example, a warehouse that maintains data on the previous twelve business quarters may periodically purge its data each year, or even each quarter.

Data warehouses must also be designed with full consideration of the environment in which they will reside. Important design considerations include the following:

- Usage projections
- The fit of the data model
- Characteristics of available sources
- Design of the meta-data component
- Modular component design
- Design for manageability and change
- Considerations of distributed and parallel architecture

We discuss each of these in turn. Warehouse design is initially driven by usage projections; that is, by expectations about who will use the warehouse and how they will use it. Choice of a data model to support this usage is a key initial decision. Usage projections and the characteristics of the warehouse's data sources are both taken into account. Modular design is a practical necessity to allow the warehouse to evolve with the organization and its information environment. Additionally, a well-built data warehouse must be designed for maintainability, enabling the warehouse managers to plan for and manage change effectively while providing optimal support to users.

You may recall the term *meta-data* from Chapter 1; meta-data was defined as the description of a database; this description includes the database's schema definition. The **meta-data repository** is a key data warehouse component. The meta-data repository includes both technical and business meta-data. The first, **technical meta-data**, covers details of acquisition, processing, storage structures, data descriptions, warehouse operations and maintenance, and access support functionality. The second, **business meta-data**, includes the relevant business rules and organizational details supporting the warehouse.

The architecture of the organization's distributed computing environment is a major determining characteristic for the design of the warehouse. There are two basic distributed architectures: the distributed warehouse and the federated warehouse. For a **distributed warehouse**, all the issues of distributed databases are relevant; for example, replication, partitioning, communications, and consistency concerns. A distributed architecture can provide benefits particularly important to warehouse performance, such as improved load balancing, scalability of performance, and higher availability. A single replicated meta-data repository would reside at each distribution site. The idea of the **federated warehouse** is like that of the federated database: a decentralized confederation of autonomous data warehouses, each with its own meta-data repository. Given the magnitude of the challenge inherent to data warehouses, it is likely that such federations will consist of smaller scale components, such as data marts.

Businesses are becoming dissatisfied with the traditional data warehousing techniques and technologies. New analytic requirements are driving new analytic appliances; examples include Netezza of IBM, Greenplum of EMC, Hana of SAP, and

ParAccel of Tableau Software. Big data analytics have driven Hadoop and other specialized databases such as graph and key-value stores into the next generation of data warehousing (see Chapter 25 for a discussion of big data technology based on Hadoop). Data virtualization platforms such as the one from Cisco<sup>4</sup> will enable such logical data warehouses to be built in the future.

## 29.5 Typical Functionality of a Data Warehouse

Data warehouses exist to facilitate complex, data-intensive, and frequent ad hoc queries. Accordingly, data warehouses must provide far greater and more efficient query support than is demanded of transactional databases. The data warehouse access component supports enhanced spreadsheet functionality, efficient query processing, structured queries, ad hoc queries, data mining, and materialized views. In particular, enhanced spreadsheet functionality includes support for state-of-the-art spreadsheet applications (for example, MS Excel) as well as for OLAP applications programs. These enhanced spreadsheet products offer preprogrammed functionalities such as the following:

- **Roll-up (also drill-up).** Data is summarized with increasing generalization (for example, weekly to quarterly to annually).
- **Drill-down.** Increasing levels of detail are revealed (the complement of roll-up).
- **Pivot.** Cross tabulation (also referred to as *rotation*) is performed.
- **Slice and dice.** Projection operations are performed on the dimensions.
- **Sorting.** Data is sorted by ordinal value.
- **Selection.** Data is filtered by value or range.
- **Derived (computed) attributes.** Attributes are computed by operations on stored and derived values.

Because data warehouses are free from the restrictions of the transactional environment, there is an increased efficiency in query processing. Among the tools and techniques used are query transformation; index intersection and union; special **ROLAP** (relational OLAP) and **MOLAP** (multidimensional OLAP) functions; SQL extensions; advanced join methods; and intelligent scanning (as in piggy-backing multiple queries).

There is also a **HOLAP** (hybrid OLAP) option available that combines both ROLAP and MOLAP. For summary-type information, HOLAP leverages cube technology (using MOLAP) for faster performance. When detailed information is needed, HOLAP can “drill through” from the cube into the underlying relational data (which is in the ROLAP component).

---

<sup>4</sup>See the description of Cisco's Data Virtualization Platform at <http://www.compositesw.com/products-services/data-virtualization-platform/>

Improved performance has also been attained with parallel processing. Parallel server architectures include symmetric multiprocessor (SMP), cluster, and massively parallel processing (MPP), and combinations of these.

Knowledge workers and decision makers use tools ranging from parametric queries to ad hoc queries to data mining. Thus, the access component of the data warehouse must provide support for structured queries (both parametric and ad hoc). Together, these make up a managed query environment. Data mining itself uses techniques from statistical analysis and artificial intelligence. Statistical analysis can be performed by advanced spreadsheets, by sophisticated statistical analysis software, or by custom-written programs. Techniques such as lagging, moving averages, and regression analysis are also commonly employed. Artificial intelligence techniques, which may include genetic algorithms and neural networks, are used for classification and are employed to discover knowledge from the data warehouse that may be unexpected or difficult to specify in queries. (We discussed data mining in detail in Chapter 28.)

## 29.6 Data Warehouse versus Views

Some people consider data warehouses to be an extension of database views. Earlier we mentioned materialized views as one way of meeting requirements for improved access to data (see Section 7.3 for a discussion of views). Materialized views have been explored for their performance enhancement. In Section 19.2.4, we discussed how materialized views are maintained and used as a part of query optimization. Views, however, provide only a subset of the functions and capabilities of data warehouses. Views and data warehouses are similar in some aspects; for example, they both have read-only extracts from databases and they allow orientation by subject. However, data warehouses are different from views in the following ways:

- Data warehouses exist as persistent storage instead of being materialized on demand.
- Data warehouses are not just relational views; they are multidimensional views with levels of aggregation.
- Data warehouses can be indexed to optimize performance. Views cannot be indexed independent of the underlying databases.
- Data warehouses characteristically provide specific support of functionality; views cannot.
- Data warehouses provide large amounts of integrated and often temporal data, generally more than is contained in one database, whereas views are an extract of a database.
- Data warehouses bring in data from multiple sources via a complex ETL process that involves cleaning, pruning, and summarization, whereas views are an extract from a database through a predefined query.

## 29.7 Difficulties of Implementing Data Warehouses

Some significant operational issues arise with data warehousing: construction, administration, and quality control. Project management—the design, construction, and implementation of the warehouse—is an important and challenging consideration that should not be underestimated. The building of an enterprise-wide warehouse in a large organization is a major undertaking, potentially taking years from conceptualization to implementation. Because of the difficulty and amount of lead time required for such an undertaking, the widespread development and deployment of data marts may provide an attractive alternative, especially to those organizations with urgent needs for OLAP, DSS, and/or data mining support.

The administration of a data warehouse is an intensive enterprise, proportional to the size and complexity of the warehouse. An organization that attempts to administer a data warehouse must realistically understand the complex nature of its administration. Although designed for read access, a data warehouse is no more a static structure than any of its information sources. Source databases can be expected to evolve. The warehouse's schema and acquisition component must be expected to be updated to handle these evolutions.

A significant issue in data warehousing is the quality control of data. Both quality and consistency of data—especially as it relates to dimension data, which in turn affects master data management—are major concerns. Although the data passes through a cleaning function during acquisition, quality and consistency remain significant issues for the database administrator and designer alike. Melding data from heterogeneous and disparate sources is a major challenge given differences in naming, domain definitions, identification numbers, and the like. Every time a source database changes, the data warehouse administrator must consider the possible interactions with other elements of the warehouse.

Usage projections should be estimated conservatively prior to construction of the data warehouse and should be revised continually to reflect current requirements. As utilization patterns become clear and change over time, storage and access paths can be tuned to remain optimized for support of the organization's use of its warehouse. This activity should continue throughout the life of the warehouse in order to remain ahead of the demand. The warehouse should also be designed to accommodate the addition and attrition of data sources without major redesign. Sources and source data will evolve, and the warehouse must accommodate such change. Fitting the available source data into the data model of the warehouse will be a continual challenge, a task that is as much art as science. Because there is continual rapid change in technologies, both the requirements and capabilities of the warehouse will change considerably over time. Additionally, data warehousing technology itself will continue to evolve for some time, so component structures and functionalities will continually be upgraded. This certain change is an excellent motivation for fully modular design of components.



Administration of a data warehouse will require far broader skills than are needed for traditional database administration. Often, different parts of a large organization view the data differently. A team of highly skilled technical experts with overlapping areas of expertise will likely be needed, rather than a single individual. The team must also possess a thorough knowledge of the business and specifically the rules and regulations, the constraints and the policies of the enterprise. Like database administration, data warehouse administration is only partly technical; a large part of the responsibility requires working effectively with all the members of the organization who have an interest in the data warehouse. However difficult that can be at times for database administrators, it is that much more challenging for data warehouse administrators because the scope of their responsibilities is considerably broader than that faced by database administrators.

Design of the management function and selection of the management team for a database warehouse are crucial. Managing the data warehouse in a large organization will surely be a major task. Many commercial tools are available to support management functions. Effective data warehouse management will be a team function that requires a wide set of technical skills, careful coordination, and effective leadership. Just as we must prepare for the evolution of the warehouse, we must also recognize that the skills of the management team will, of necessity, evolve with it.

## 29.8 Summary

In this chapter, we surveyed the field known as data warehousing. Data warehousing can be seen as a process that requires a variety of activities to precede it. In contrast, data mining (see Chapter 28) may be thought of as an activity that draws knowledge from an existing data warehouse or other sources of data. We first introduced in Section 29.1 key concepts related to a data warehouse and defined terms such as *OLAP* and *DSS* and contrasted them with *OLTP*. We presented a general architecture of data warehousing systems. We discussed in Section 29.2 the fundamental characteristics of data warehouses and their different types. We then discussed in Section 29.3 the modeling of data in warehouses using what is popularly known as the multidimensional data model. Different types of tables and schemas were discussed. We gave an elaborate account of the processes and design considerations involved in building a data warehouse in Section 29.4. We then presented the typical special functionality associated with a data warehouse in Section 29.5. The view concept from the relational model was contrasted with the multidimensional view of data in data warehouses in Section 29.6. We finally discussed in Section 29.7 the difficulties of implementing data warehouses and the challenges of data warehouse administration.

## Review Questions

**29.1.** What is a data warehouse? How does it differ from a database?

**29.2.** Define the following terms: *OLAP* (online analytical processing), *ROLAP* (relational OLAP), *MOLAP* (multidimensional OLAP), and *DSS* (decision-support systems).



- 29.3. Describe the characteristics of a data warehouse. Divide them into the functionality of a warehouse and the advantages users derive from the warehouse.
- 29.4. What is the multidimensional data model? How is it used in data warehousing?
- 29.5. Define the following terms: *star schema*, *snowflake schema*, *fact constellation*, *data marts*.
- 29.6. What types of indexes are built for a warehouse? Illustrate the uses for each with an example.
- 29.7. Describe the steps of building a warehouse.
- 29.8. What considerations play a major role in the design of a warehouse?
- 29.9. Describe the functions a user can perform on a data warehouse, and illustrate the results of these functions on a sample multidimensional data warehouse.
- 29.10. How is the relational *view* concept similar to a data warehouse and how are they different?
- 29.11. List the difficulties in implementing a data warehouse.
- 29.12. List the ongoing issues and research problems pertaining to data warehousing.
- 29.13. What is master data management? How is it related to data warehousing?
- 29.14. What are logical data warehouses? Do an online search for the data virtualization platform from Cisco, and discuss how it will help in building a logical data warehouse?

## Selected Bibliography

Inmon (1992, 2005) is credited for giving the term wide acceptance. Codd and Salley (1993) popularized the term *online analytical processing* (OLAP) and defined a set of characteristics for data warehouses to support OLAP. Kimball (1996) is known for his contribution to the development of the data warehousing field. Mattison (1996) is one of the several books on data warehousing that gives a comprehensive analysis of techniques available in data warehouses and the strategies companies should use in deploying them. Ponniah (2010) gives a very good practical overview of the data warehouse building process from requirements collection to deployment maintenance. Jukic et al. (2013) is a good source on modeling a data warehouse. Bischoff and Alexander (1997) is a compilation of advice from experts. Chaudhuri and Dayal (1997) give an excellent tutorial on the topic, while Widom (1995) points to a number of ongoing issues and research.

# part 12

## **Additional Database Topics: Security**

This page intentionally left blank

## Database Security

This chapter discusses techniques for securing databases against a variety of threats. It also presents schemes of providing access privileges to authorized users. Some of the security threats to databases—such as SQL injection—will be presented. At the end of the chapter, we summarize how a mainstream RDBMS—specifically, the Oracle system—provides different types of security. We start in Section 30.1 with an introduction to security issues and the threats to databases, and we give an overview of the control measures that are covered in the rest of this chapter. We also comment on the relationship between data security and privacy as it applies to personal information. Section 30.2 discusses the mechanisms used to grant and revoke privileges in relational database systems and in SQL, mechanisms that are often referred to as **discretionary access control**. In Section 30.3, we present an overview of the mechanisms for enforcing multiple levels of security—a particular concern in database system security that is known as **mandatory access control**. Section 30.3 also introduces the more recently developed strategies of **role-based access control**, and label-based and row-based security. Section 30.3 also provides a brief discussion of XML access control. Section 30.4 discusses a major threat to databases—SQL injection—and discusses some of the proposed preventive measures against it. Section 30.5 briefly discusses the security problem in statistical databases. Section 30.6 introduces the topic of flow control and mentions problems associated with covert channels. Section 30.7 provides a brief summary of encryption and symmetric key and asymmetric (public) key infrastructure schemes. It also discusses digital certificates. Section 30.8 introduces privacy-preserving techniques, and Section 30.9 presents the current challenges to database security. In Section 30.10, we discuss Oracle label-based security. Finally, Section 30.11 summarizes the chapter. Readers who are interested only in basic database security mechanisms will find it sufficient to cover the material in Sections 30.1 and 30.2.

## 30.1 Introduction to Database Security Issues<sup>1</sup>

### 30.1.1 Types of Security

Database security is a broad area that addresses many issues, including the following:

- Various legal and ethical issues regarding the right to access certain information—for example, some information may be deemed to be private and cannot be accessed legally by unauthorized organizations or persons. In the United States, there are numerous laws governing privacy of information.
- Policy issues at the governmental, institutional, or corporate level regarding what kinds of information should not be made publicly available—for example, credit ratings and personal medical records.
- System-related issues such as the *system levels* at which various security functions should be enforced—for example, whether a security function should be handled at the physical hardware level, the operating system level, or the DBMS level.
- The need in some organizations to identify multiple *security levels* and to categorize the data and users based on these classifications—for example, top secret, secret, confidential, and unclassified. The security policy of the organization with respect to permitting access to various classifications of data must be enforced.

**Threats to Databases.** Threats to databases can result in the loss or degradation of some or all of the following commonly accepted security goals: integrity, availability, and confidentiality.

- **Loss of integrity.** Database integrity refers to the requirement that information be protected from improper modification. Modification of data includes creating, inserting, and updating data; changing the status of data; and deleting data. Integrity is lost if unauthorized changes are made to the data by either intentional or accidental acts. If the loss of system or data integrity is not corrected, continued use of the contaminated system or corrupted data could result in inaccuracy, fraud, or erroneous decisions.
- **Loss of availability.** *Database availability* refers to making objects available to a human user or a program who/which has a legitimate right to those data objects. *Loss of availability* occurs when the user or program cannot access these objects.
- **Loss of confidentiality.** Database confidentiality refers to the protection of data from unauthorized disclosure. The impact of unauthorized disclosure of confidential information can range from violation of the Data Privacy Act to the jeopardization of national security. Unauthorized, unanticipated, or unintentional disclosure could result in loss of public confidence, embarrassment, or legal action against the organization.

---

<sup>1</sup>The substantial contributions of Fariborz Farahmand, Bharath Rengarajan, and Frank Rietta to this and subsequent sections of this chapter is much appreciated.

**Database Security: Not an Isolated Concern.** When considering the threats facing databases, it is important to remember that the database management system alone cannot be responsible for maintaining the confidentiality, integrity, and availability of the data. Rather, the database works as part of a network of services, including applications, Web servers, firewalls, SSL terminators, and security monitoring systems. Because security of an overall system is only as strong as its weakest link, a database may be compromised even if it would have been perfectly secure on its own merits.

To protect databases against the threats discussed above, it is common to implement *four kinds of control measures*: access control, inference control, flow control, and encryption. We discuss each of these in this chapter.

In a multiuser database system, the DBMS must provide techniques to enable certain users or user groups to access selected portions of a database without gaining access to the rest of the database. This is particularly important when a large integrated database is to be used by many different users within the same organization. For example, sensitive information such as employee salaries or performance reviews should be kept confidential from most of the database system's users. A DBMS typically includes a **database security and authorization subsystem** that is responsible for ensuring the security of portions of a database against unauthorized access. It is now customary to refer to two types of database security mechanisms:

- **Discretionary security mechanisms.** These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).
- **Mandatory security mechanisms.** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization. For example, a typical security policy is to permit users at a certain classification (or clearance) level to see only the data items classified at the user's own (or lower) classification level. An extension of this is *role-based security*, which enforces policies and privileges based on the concept of organizational roles. (See Section 30.4.2 for role based access control.)

We discuss discretionary security in Section 30.2 and mandatory and role-based security in Section 30.3.

### 30.1.2 Control Measures

Four main control measures are used to provide security of data in databases:

- Access control
- Inference control
- Flow control
- Data encryption

A security problem common to computer systems is that of preventing unauthorized persons from accessing the system itself, either to obtain information or to make malicious changes in a portion of the database. The security mechanism of a DBMS must include provisions for restricting access to the database system as a whole. This function, called **access control**, is handled by creating user accounts and passwords to control the login process by the DBMS. We discuss access control techniques in Section 30.1.3.

**Statistical databases** are used to provide statistical information or summaries of values based on various criteria. For example, a database for population statistics may provide statistics based on age groups, income levels, household size, education levels, and other criteria. Statistical database users such as government statisticians or market research firms are allowed to access the database to retrieve statistical information about a population but not to access the detailed confidential information about specific individuals. Security for statistical databases must ensure that information about individuals cannot be accessed. It is sometimes possible to deduce or infer certain facts concerning individuals from queries that involve only summary statistics on groups; consequently, this must not be permitted either. This problem, called **statistical database security**, is discussed briefly in Section 30.4. The corresponding control measures are called **inference control** measures.

Another security issue is that of **flow control**, which prevents information from flowing in such a way that it reaches unauthorized users. Flow control is discussed in Section 30.6. **Covert channels** are pathways on which information flows implicitly in ways that violate the security policy of an organization. We briefly discuss some issues related to covert channels in Section 30.6.1.

A final control measure is **data encryption**, which is used to protect sensitive data (such as credit card numbers) that is transmitted via some type of communications network. Encryption can be used to provide additional protection for sensitive portions of a database as well. The data is **encoded** using some coding algorithm. An unauthorized user who accesses encoded data will have difficulty deciphering it, but authorized users are given decoding or decrypting algorithms (or keys) to decipher the data. Encrypting techniques that are very difficult to decode without a key have been developed for military applications. However, encrypted database records are used today in both private organizations and governmental and military applications. In fact, state and federal laws prescribe encryption for any system that deals with legally protected personal information. For example, according to Georgia Law (OCGA 10-1-911):

“Personal information” means an individual’s first name or first initial and last name in combination with any one or more of the following data elements, when either the name or the data elements are not encrypted or redacted:

- ❑ Social security number;
- ❑ Driver’s license number or state identification card number;

- Account number, credit card number, or debit card number, if circumstances exist wherein such a number could be used without additional identifying information, access codes, or passwords;
- Account passwords or personal identification numbers or other access codes

Because laws defining what constitutes personal information vary from state to state, systems must protect individuals' privacy and enforce privacy measures adequately. Discretionary access control (see Section 30.2) alone may not suffice. Section 30.7 briefly discusses encryption techniques, including popular techniques such as public key encryption (which is heavily used to support Web-based transactions against databases) and digital signatures (which are used in personal communications).

A comprehensive discussion of security in computer systems and databases is outside the scope of this text. We give only a brief overview of database security techniques here. Network- and communication-based security is also a vast topic that we do not cover. For a comprehensive discussion, the interested reader can refer to several of the references discussed in the Selected Bibliography at the end of this chapter.

### 30.1.3 Database Security and the DBA

As we discussed in Chapter 1, the database administrator (DBA) is the central authority for managing a database system. The DBA's responsibilities include granting privileges to users who need to use the system and classifying users and data in accordance with the policy of the organization. The DBA has a **DBA account** in the DBMS, sometimes called a **system** or **superuser account**, which provides powerful capabilities that are not made available to regular database accounts and users.<sup>2</sup> DBA-privileged commands include commands for granting and revoking privileges to individual accounts, users, or user groups and for performing the following types of actions:

1. **Account creation.** This action creates a new account and password for a user or a group of users to enable access to the DBMS.
2. **Privilege granting.** This action permits the DBA to grant certain privileges to certain accounts.
3. **Privilege revocation.** This action permits the DBA to revoke (cancel) certain privileges that were previously given to certain accounts.
4. **Security level assignment.** This action consists of assigning user accounts to the appropriate security clearance level.

The DBA is responsible for the overall security of the database system. Action 1 in the preceding list is used to control access to the DBMS as a whole, whereas actions 2 and 3 are used to control *discretionary* database authorization, and action 4 is used to control *mandatory* authorization.

---

<sup>2</sup>This account is similar to the *root* or *superuser* accounts that are given to computer system administrators and that allow access to restricted operating system commands.



### 30.1.4 Access Control, User Accounts, and Database Audits

Whenever a person or a group of persons needs to access a database system, the individual or group must first apply for a user account. The DBA will then create a new **account number** and **password** for the user if there is a legitimate need to access the database. The user must **log in** to the DBMS by entering the account number and password whenever database access is needed. The DBMS checks that the account number and password are valid; if they are, the user is permitted to use the DBMS and to access the database. Application programs can also be considered users and are required to log in to the database (see Chapter 10).

It is straightforward to keep track of database users and their accounts and passwords by creating an encrypted table or file with two fields: AccountNumber and Password. This table can easily be maintained by the DBMS. Whenever a new account is created, a new record is inserted into the table. When an account is canceled, the corresponding record must be deleted from the table.

The database system must also keep track of all operations on the database that are applied by a certain user throughout each **login session**, which consists of the sequence of database interactions that a user performs from the time of logging in to the time of logging off. When a user logs in, the DBMS can record the user's account number and associate it with the computer or device from which the user logged in. All operations applied from that computer or device are attributed to the user's account until the user logs off. It is particularly important to keep track of update operations that are applied to the database so that, if the database is tampered with, the DBA can determine which user did the tampering.

To keep a record of all updates applied to the database and of particular users who applied each update, we can modify the *system log*. Recall from Chapters 20 and 22 that the **system log** includes an entry for each operation applied to the database that may be required for recovery from a transaction failure or system crash. We can expand the log entries so that they also include the account number of the user and the online computer or device ID that applied each operation recorded in the log. If any tampering with the database is suspected, a **database audit** is performed, which consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period. When an illegal or unauthorized operation is found, the DBA can determine the account number used to perform the operation. Database audits are particularly important for sensitive databases that are updated by many transactions and users, such as a banking database that can be updated by thousands of bank tellers. A database log that is used mainly for security purposes serves as an **audit trail**.

### 30.1.5 Sensitive Data and Types of Disclosures

**Sensitivity of data** is a measure of the importance assigned to the data by its owner for the purpose of denoting its need for protection. Some databases contain only sensitive data whereas other databases may contain no sensitive data at all. Handling databases that fall at these two extremes is relatively easy because

such databases can be covered by access control, which is explained in the next section. The situation becomes tricky when some of the data is sensitive whereas other data is not.

Several factors can cause data to be classified as sensitive:

1. **Inherently sensitive.** The value of the data itself may be so revealing or confidential that it becomes sensitive—for example, a person's salary or who a patient has HIV/AIDS.
2. **From a sensitive source.** The source of the data may indicate a need for secrecy—for example, an informer whose identity must be kept secret.
3. **Declared sensitive.** The owner of the data may have explicitly declared it as sensitive.
4. **A sensitive attribute or sensitive record.** The particular attribute or record may have been declared sensitive—for example, the salary attribute of an employee or the salary history record in a personnel database.
5. **Sensitive in relation to previously disclosed data.** Some data may not be sensitive by itself but will become sensitive in the presence of some other data—for example, the exact latitude and longitude information for a location where some previously recorded event happened that was later deemed sensitive.

It is the responsibility of the database administrator and security administrator to collectively enforce the security policies of an organization. This dictates whether access should or should not be permitted to a certain database attribute (also known as a *table column* or a *data element*) for individual users or for categories of users. Several factors must be considered before deciding whether it is safe to reveal the data. The three most important factors are data availability, access acceptability, and authenticity assurance.

1. **Data availability.** If a user is updating a field, then this field becomes inaccessible and other users should not be able to view this data. This blocking is only temporary and only to ensure that no user sees any inaccurate data. This is typically handled by the concurrency control mechanism (see Chapter 21).
2. **Access acceptability.** Data should only be revealed to authorized users. A database administrator may also deny access to a user request even if the request does not directly access a sensitive data item, on the grounds that the requested data may reveal information about the sensitive data that the user is not authorized to have.
3. **Authenticity assurance.** Before granting access, certain external characteristics about the user may also be considered. For example, a user may only be permitted access during working hours. The system may track previous queries to ensure that a combination of queries does not reveal sensitive data. The latter is particularly relevant to statistical database queries (see Section 30.5).

The term *precision*, when used in the security area, refers to allowing as much as possible of the data to be available, subject to protecting exactly the subset of data that is sensitive. The definitions of *security* versus *precision* are as follows:

- **Security:** Means of ensuring that data is kept safe from corruption and that access to it is suitably controlled. To provide security means to disclose only nonsensitive data and to reject any query that references a sensitive field.
- **Precision:** To protect all sensitive data while disclosing or making available as much nonsensitive data as possible. Note that this definition of *precision* is not related to the precision of information retrieval defined in Section 27.6.1.

The ideal combination is to maintain perfect security with maximum precision. If we want to maintain security, precision must be sacrificed to some degree. Hence there is typically a tradeoff between security and precision.

### 30.1.6 Relationship between Information Security and Information Privacy

The rapid advancement of the use of information technology (IT) in industry, government, and academia raises challenging questions and problems regarding the protection and use of personal information. Questions of *who* has *what* rights to information about individuals for *which* purposes become more important as we move toward a world in which it is technically possible to know just about anything about anyone.

Deciding how to design privacy considerations in technology for the future includes philosophical, legal, and practical dimensions. There is a considerable overlap between issues related to access to resources (security) and issues related to appropriate use of information (privacy). We now define the difference between *security* and *privacy*.

**Security** in information technology refers to many aspects of protecting a system from unauthorized use, including authentication of users, information encryption, access control, firewall policies, and intrusion detection. For our purposes here, we will limit our treatment of security to the concepts associated with how well a system can protect access to information it contains. The concept of **privacy** goes beyond security. Privacy examines how well the use of personal information that the system acquires about a user conforms to the explicit or implicit assumptions regarding that use. From an end user perspective, privacy can be considered from two different perspectives: *preventing storage* of personal information versus *ensuring appropriate use* of personal information.

For the purposes of this chapter, a simple but useful definition of **privacy** is *the ability of individuals to control the terms under which their personal information is acquired and used*. In summary, security involves technology to ensure that information is appropriately protected. Security is a required building block for privacy. Privacy involves mechanisms to support compliance with some basic principles and other explicitly stated policies. One basic principle is that people should be informed

about information collection, told in advance what will be done with their information, and given a reasonable opportunity to approve or disapprove of such use of the information. A related concept, **trust**, relates to both security and privacy and is seen as increasing when it is perceived that both security and privacy are provided for.

## 30.2 Discretionary Access Control Based on Granting and Revoking Privileges

The typical method of enforcing **discretionary access control** in a database system is based on the granting and revoking of **privileges**. Let us consider privileges in the context of a relational DBMS. In particular, we will discuss a system of privileges somewhat similar to the one originally developed for the SQL language (see Chapters 7 and 8). Many current relational DBMSs use some variation of this technique. The main idea is to include statements in the query language that allow the DBA and selected users to grant and revoke privileges.

### 30.2.1 Types of Discretionary Privileges

In SQL2 and later versions,<sup>3</sup> the concept of an **authorization identifier** is used to refer, roughly speaking, to a user account (or group of user accounts). For simplicity, we will use the words *user* or *account* interchangeably in place of *authorization identifier*. The DBMS must provide selective access to each relation in the database based on specific accounts. Operations may also be controlled; thus, having an account does not necessarily entitle the account holder to all the functionality provided by the DBMS. Informally, there are two levels for assigning privileges to use the database system:

- **The account level.** At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
- **The relation (or table) level.** At this level, the DBA can control the privilege to access each individual relation or view in the database.

The privileges at the **account level** apply to the capabilities provided to the account itself and can include the CREATE SCHEMA or CREATE TABLE privilege, to create a schema or base relation; the CREATE VIEW privilege; the ALTER privilege, to apply schema changes such as adding or removing attributes from relations; the DROP privilege, to delete relations or views; the MODIFY privilege, to insert, delete, or update tuples; and the SELECT privilege, to retrieve information from the database by using a SELECT query. Notice that these account privileges apply to the account in general. If a certain account does not have the CREATE TABLE privilege, no relations can be created from that account. Account-level privileges *are not* defined as part of SQL2; they are left to the DBMS implementers to define. In earlier versions of SQL, a CREATETAB privilege existed to give an account the privilege to create tables (relations).

---

<sup>3</sup>Discretionary privileges were incorporated into SQL2 and are applicable to later versions of SQL.

The second level of privileges applies to the **relation level**, which includes base relations and virtual (view) relations. These privileges *are* defined for SQL2. In the following discussion, the term *relation* may refer either to a base relation or to a view, unless we explicitly specify one or the other. Privileges at the relation level specify for each user the individual relations on which each type of command can be applied. Some privileges also refer to individual columns (attributes) of relations. SQL2 commands provide privileges at the *relation and attribute level only*. Although this distinction is general, it makes it difficult to create accounts with limited privileges. The granting and revoking of privileges generally follow an authorization model for discretionary privileges known as the **access matrix model**, where the rows of a matrix  $M$  represent *subjects* (users, accounts, programs) and the columns represent *objects* (relations, records, columns, views, operations). Each position  $M(i, j)$  in the matrix represents the types of privileges (read, write, update) that subject  $i$  holds on object  $j$ .

To control the granting and revoking of relation privileges, each relation  $R$  in a database is assigned an **owner account**, which is typically the account that was used when the relation was created in the first place. The owner of a relation is given *all* privileges on that relation. In SQL2, the DBA can assign an owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the CREATE SCHEMA command (see Section 7.1.1). The owner account holder can pass privileges on any of the owned relations to other users by **granting** privileges to their accounts. In SQL, the following types of privileges can be granted on each individual relation  $R$ :

- **SELECT (retrieval or read) privilege on  $R$ .** Gives the account retrieval privilege. In SQL, this gives the account the privilege to use the SELECT statement to retrieve tuples from  $R$ .
- **Modification privileges on  $R$ .** This gives the account the capability to modify the tuples of  $R$ . In SQL, this includes three privileges: UPDATE, DELETE, and INSERT. These correspond to the three SQL commands (see Section 7.4) for modifying a table  $R$ . Additionally, both the INSERT and UPDATE privileges can specify that only certain attributes of  $R$  can be modified by the account.
- **References privilege on  $R$ .** This gives the account the capability to *reference* (or refer to) a relation  $R$  when specifying integrity constraints. This privilege can also be restricted to specific attributes of  $R$ .

Notice that to create a view, the account must have the SELECT privilege on *all relations* involved in the view definition in order to specify the query that corresponds to the view.

### 30.2.2 Specifying Privileges through the Use of Views

The mechanism of **views** is an important *discretionary authorization mechanism* in its own right. For example, if the owner  $A$  of a relation  $R$  wants another account  $B$  to be able to retrieve only some fields of  $R$ , then  $A$  can create a view  $V$  of  $R$  that

includes only those attributes and then grant SELECT on  $V$  to  $B$ . The same applies to limiting  $B$  to retrieving only certain tuples of  $R$ ; a view  $V'$  can be created by defining the view by means of a query that selects only those tuples from  $R$  that  $A$  wants to allow  $B$  to access. We will illustrate this discussion with the example given in Section 30.2.5.

### 30.2.3 Revoking of Privileges

In some cases, it is desirable to grant a privilege to a user temporarily. For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for **revoking** privileges is needed. In SQL, a REVOKE command is included for the purpose of canceling privileges. We will see how the REVOKE command is used in the example in Section 30.2.5.

### 30.2.4 Propagation of Privileges Using the GRANT OPTION

Whenever the owner  $A$  of a relation  $R$  grants a privilege on  $R$  to another account  $B$ , the privilege can be given to  $B$  *with* or *without* the **GRANT OPTION**. If the GRANT OPTION is given, this means that  $B$  can also grant that privilege on  $R$  to other accounts. Suppose that  $B$  is given the GRANT OPTION by  $A$  and that  $B$  then grants the privilege on  $R$  to a third account  $C$ , also with the GRANT OPTION. In this way, privileges on  $R$  can **propagate** to other accounts without the knowledge of the owner of  $R$ . If the owner account  $A$  now revokes the privilege granted to  $B$ , all the privileges that  $B$  propagated based on that privilege *should automatically be revoked* by the system.

It is possible for a user to receive a certain privilege from two or more sources. For example,  $A_4$  may receive a certain UPDATE  $R$  privilege from *both*  $A_2$  and  $A_3$ . In such a case, if  $A_2$  revokes this privilege from  $A_4$ ,  $A_4$  will still continue to have the privilege by virtue of having been granted it from  $A_3$ . If  $A_3$  later revokes the privilege from  $A_4$ ,  $A_4$  totally loses the privilege. Hence, a DBMS that allows propagation of privileges must keep track of how all the privileges were granted in the form of some internal log so that revoking of privileges can be done correctly and completely.

### 30.2.5 An Example to Illustrate Granting and Revoking of Privileges

Suppose that the DBA creates four accounts— $A_1$ ,  $A_2$ ,  $A_3$ , and  $A_4$ —and wants only  $A_1$  to be able to create base relations. To do this, the DBA must issue the following GRANT command in SQL:

```
GRANT CREATETAB TO A1;
```

The CREATETAB (create table) privilege gives account  $A_1$  the capability to create new database tables (base relations) and is hence an *account privilege*. This privilege was part of earlier versions of SQL but is now left to each individual system

implementation to define. Note that A1 , A2, and so forth may be individuals, like John in IT department or Mary in marketing; but they may also be applications or programs that want to access a database.

In SQL2, the same effect can be accomplished by having the DBA issue a CREATE SCHEMA command, as follows:

**CREATE SCHEMA EXAMPLE AUTHORIZATION A1;**

User account A1 can now create tables under the schema called EXAMPLE. To continue our example, suppose that A1 creates the two base relations EMPLOYEE and DEPARTMENT shown in Figure 30.1; A1 is then the **owner** of these two relations and hence has *all the relation privileges* on each of them.

Next, suppose that account A1 wants to grant to account A2 the privilege to insert and delete tuples in both of these relations. However, A1 does not want A2 to be able to propagate these privileges to additional accounts. A1 can issue the following command:

**GRANT INSERT, DELETE ON EMPLOYEE, DEPARTMENT TO A2;**

Notice that the owner account A1 of a relation automatically has the GRANT OPTION, allowing it to grant privileges on the relation to other accounts. However, account A2 cannot grant INSERT and DELETE privileges on the EMPLOYEE and DEPARTMENT tables because A2 was not given the GRANT OPTION in the preceding command.

Next, suppose that A1 wants to allow account A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts. A1 can issue the following command:

**GRANT SELECT ON EMPLOYEE, DEPARTMENT TO A3 WITH GRANT OPTION;**

The clause WITH GRANT OPTION means that A3 can now propagate the privilege to other accounts by using GRANT. For example, A3 can grant the SELECT privilege on the EMPLOYEE relation to A4 by issuing the following command:

**GRANT SELECT ON EMPLOYEE TO A4;**

Notice that A4 cannot propagate the SELECT privilege to other accounts because the GRANT OPTION was not given to A4.

Now suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 then can issue this command:

**REVOKE SELECT ON EMPLOYEE FROM A3;**

**Figure 30.1**  
Schemas for the two relations  
EMPLOYEE and DEPARTMENT.

EMPLOYEE						
Name	<u>Ssn</u>	Bdate	Address	Sex	Salary	Dno

DEPARTMENT		
<u>Dnumber</u>	Dname	Mgr_ssn



The DBMS must now revoke the SELECT privilege on EMPLOYEE from A3, and it must also *automatically revoke* the SELECT privilege on EMPLOYEE from A4. This is because A3 granted that privilege to A4, but A3 does not have the privilege any more.

Next, suppose that A1 wants to give back to A3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege. The limitation is to retrieve only the Name, Bdate, and Address attributes and only for the tuples with Dno = 5. A1 then can create the following view:

```
CREATE VIEW A3EMPLOYEE AS
SELECT Name, Bdate, Address
FROM EMPLOYEE
WHERE Dno = 5;
```

After the view is created, A1 can grant SELECT on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3 WITH GRANT OPTION;
```

Finally, suppose that A1 wants to allow A4 to update only the Salary attribute of EMPLOYEE; A1 can then issue the following command:

```
GRANT UPDATE ON EMPLOYEE (Salary) TO A4;
```

The UPDATE and INSERT privileges can specify particular attributes that may be updated or inserted in a relation. Other privileges (SELECT, DELETE) are not attribute specific, because this specificity can easily be controlled by creating the appropriate views that include only the desired attributes and granting the corresponding privileges on the views. However, because updating views is not always possible (see Chapter 8), the UPDATE and INSERT privileges are given the option to specify the particular attributes of a base relation that may be updated.

### 30.2.6 Specifying Limits on Propagation of Privileges

Techniques to limit the propagation of privileges have been developed, although they have not yet been implemented in most DBMSs and *are not a part of SQL*. Limiting **horizontal propagation** to an integer number  $i$  means that an account  $B$  given the GRANT OPTION can grant the privilege to at most  $i$  other accounts. **Vertical propagation** is more complicated; it limits the depth of the granting of privileges. Granting a privilege with a vertical propagation of zero is equivalent to granting the privilege with *no* GRANT OPTION. If account  $A$  grants a privilege to account  $B$  with the vertical propagation set to an integer number  $j > 0$ , this means that the account  $B$  has the GRANT OPTION on that privilege, but  $B$  can grant the privilege to other accounts only with a vertical propagation *less than*  $j$ . In effect, vertical propagation limits the sequence of GRANT OPTIONS that can be given from one account to the next based on a single original grant of the privilege.

We briefly illustrate horizontal and vertical propagation limits—which are *not available* currently in SQL or other relational systems—with an example. Suppose



that A1 grants SELECT to A2 on the EMPLOYEE relation with horizontal propagation equal to 1 and vertical propagation equal to 2. A2 can then grant SELECT to at most one account because the horizontal propagation limitation is set to 1. Additionally, A2 cannot grant the privilege to another account except with vertical propagation set to 0 (no GRANT OPTION) or 1; this is because A2 must reduce the vertical propagation by at least 1 when passing the privilege to others. In addition, the horizontal propagation must be less than or equal to the originally granted horizontal propagation. For example, if account A grants a privilege to account B with the horizontal propagation set to an integer number  $j > 0$ , this means that B can grant the privilege to other accounts only with a horizontal propagation *less than or equal to*  $j$ . As this example shows, horizontal and vertical propagation techniques are designed to limit the depth and breadth of propagation of privileges.

### 30.3 Mandatory Access Control and Role-Based Access Control for Multilevel Security

The discretionary access control technique of granting and revoking privileges on relations has traditionally been the main security mechanism for relational database systems. This is an all-or-nothing method: A user either has or does not have a certain privilege. In many applications, an *additional security policy* is needed that classifies data and users based on security classes. This approach, known as **mandatory access control (MAC)**, would typically be *combined* with the discretionary access control mechanisms described in Section 30.2. It is important to note that most mainstream RDBMSs currently provide mechanisms only for discretionary access control. However, the need for multilevel security exists in government, military, and intelligence applications, as well as in many industrial and corporate applications. Because of the overriding concerns for privacy, in many systems the levels are determined by who has what access to what private information (also called personally identifiable information). Some DBMS vendors—for example, Oracle—have released special versions of their RDBMSs that incorporate mandatory access control for government use.

Typical **security classes** are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest. Other more complex security classification schemes exist, in which the security classes are organized in a lattice. For simplicity, we will use the system with four security classification levels, where  $TS \geq S \geq C \geq U$ , to illustrate our discussion. The commonly used model for multilevel security, known as the *Bell-LaPadula model*,<sup>4</sup> classifies each **subject** (user, account, program) and **object** (relation, tuple, column, view, operation) into one of the security classifications TS, S, C, or U. We will refer to the **clearance** (classification) of a subject  $S$  as **class(S)** and to the **classification** of an object  $O$  as **class(O)**. Two restrictions are enforced on data access based on the subject/object classifications:

---

<sup>4</sup>Bell and La Padulla (1976) was a MITRE technical report on secure computer systems in Multics.

1. A subject  $S$  is not allowed read access to an object  $O$  unless  $\text{class}(S) \geq \text{class}(O)$ . This is known as the **simple security property**.
2. A subject  $S$  is not allowed to write an object  $O$  unless  $\text{class}(S) \leq \text{class}(O)$ . This is known as the **star property** (or  $*$ -property).

The first restriction is intuitive and enforces the obvious rule that no subject can read an object whose security classification is higher than the subject's security clearance. The second restriction is less intuitive. It prohibits a subject from writing an object at a lower security classification than the subject's security clearance. Violation of this rule would allow information to flow from higher to lower classifications, which violates a basic tenet of multilevel security. For example, a user (subject) with TS clearance may make a copy of an object with classification TS and then write it back as a new object with classification U, thus making it visible throughout the system.

To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects. Hence, each attribute  $A$  is associated with a **classification attribute**  $C$  in the schema, and each attribute value in a tuple is associated with a corresponding security classification. In addition, in some models, a **tuple classification** attribute  $TC$  is added to the relation attributes to provide a classification for each tuple as a whole. The model we describe here is known as the *multilevel model*, because it allows classifications at multiple security levels. A **multilevel relation** schema  $R$  with  $n$  attributes would be represented as:

$$R(A_1, C_1, A_2, C_2, \dots, A_n, C_n, TC)$$

where each  $C_i$  represents the *classification attribute* associated with attribute  $A_i$ .

The value of the tuple classification attribute  $TC$  in each tuple  $t$ —which is the *highest* of all attribute classification values within  $t$ —provides a general classification for the tuple itself. Each attribute classification  $C_i$  provides a finer security classification for each attribute value within the tuple. The value of  $TC$  in each tuple  $t$  is the *highest* of all attribute classification values  $C_i$  within  $t$ .

The **apparent key** of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation. A multilevel relation will appear to contain different data to subjects (users) with different clearance levels. In some cases, it is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuples at a lower-level classification through a process known as **filtering**. In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the *apparent key*. This leads to the concept of **polyinstantiation**,<sup>5</sup> where several tuples can have the same apparent key value but have different attribute values for users at different clearance levels.

We illustrate these concepts with the simple example of a multilevel relation shown in Figure 30.2(a), where we display the classification attribute values next to each

---

<sup>5</sup>This is similar to the notion of having multiple versions in the database that represent the same real-world object.

**(a) EMPLOYEE**

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Brown C	80000 S	Good C	S

**(b) EMPLOYEE**

Name	Salary	JobPerformance	TC
Smith U	40000 C	NULL C	C
Brown C	NULL C	Good C	C

**Figure 30.2**

A multilevel relation to illustrate multilevel security. (a) The original EMPLOYEE tuples. (b) Appearance of EMPLOYEE after filtering for classification C users. (c) Appearance of EMPLOYEE after filtering for classification U users. (d) Polyinstantiation of the Smith tuple.

**(c) EMPLOYEE**

Name	Salary	JobPerformance	TC
Smith U	NULL U	NULL U	U

**(d) EMPLOYEE**

Name	Salary	JobPerformance	TC
Smith U	40000 C	Fair S	S
Smith U	40000 C	Excellent C	C
Brown C	80000 S	Good C	S

attribute's value. Assume that the Name attribute is the apparent key, and consider the query **SELECT \* FROM EMPLOYEE**. A user with security clearance S would see the same relation shown in Figure 30.2(a), since all tuple classifications are less than or equal to S. However, a user with security clearance C would not be allowed to see the values for Salary of 'Brown' and Job\_performance of 'Smith', since they have higher classification. The tuples would be **filtered** to appear as shown in Figure 30.2(b), with Salary and Job\_performance *appearing as null*. For a user with security clearance U, the filtering allows only the Name attribute of 'Smith' to appear, with all the other attributes appearing as null (Figure 30.2(c)). Thus, filtering introduces null values for attribute values whose security classification is higher than the user's security clearance.

In general, the **entity integrity** rule for multilevel relations states that all attributes that are members of the apparent key must not be null and must have the *same* security classification within each individual tuple. Additionally, all other attribute values in the tuple must have a security classification greater than or equal to that of the apparent key. This constraint ensures that a user can see the key if the user is permitted to see any part of the tuple. Other integrity rules, called **null integrity** and **interinstance integrity**, informally ensure that if a tuple value at some security level can be filtered (derived) from a higher-classified tuple, then it is sufficient to store the higher-classified tuple in the multilevel relation.

To illustrate polyinstantiation further, suppose that a user with security clearance *C* tries to update the value of *Job\_performance* of ‘Smith’ in Figure 30.2 to ‘Excellent’; this corresponds to the following SQL update being submitted by that user:

```
UPDATE EMPLOYEE
SET      Job_performance = ‘Excellent’
WHERE    Name = ‘Smith’;
```

Since the view provided to users with security clearance *C* (see Figure 30.2(b)) permits such an update, the system should not reject it; otherwise, the user could *infer* that some nonnull value exists for the *Job\_performance* attribute of ‘Smith’ rather than the null value that appears. This is an example of inferring information through what is known as a **covert channel**, which should not be permitted in highly secure systems (see Section 30.6.1). However, the user should not be allowed to overwrite the existing value of *Job\_performance* at the higher classification level. The solution is to create a **polyinstantiation** for the ‘Smith’ tuple at the lower classification level *C*, as shown in Figure 30.2(d). This is necessary since the new tuple cannot be filtered from the existing tuple at classification *S*.

The basic update operations of the relational model (INSERT, DELETE, UPDATE) must be modified to handle this and similar situations, but this aspect of the problem is outside the scope of our presentation. We refer the interested reader to the Selected Bibliography at the end of this chapter for further details.

### 30.3.1 Comparing Discretionary Access Control and Mandatory Access Control

Discretionary access control (DAC) policies are characterized by a high degree of flexibility, which makes them suitable for a large variety of application domains. The main drawback of DAC models is their vulnerability to malicious attacks, such as Trojan horses embedded in application programs. The reason for this vulnerability is that discretionary authorization models do not impose any control on how information is propagated and used once it has been accessed by users authorized to do so. By contrast, mandatory policies ensure a high degree of protection—in a way, they prevent any illegal flow of information. Therefore, they are suitable for military and high-security types of applications, which require a higher degree of protection. However, mandatory policies have the drawback of being too rigid in that they require a strict classification of subjects and objects into security levels, and therefore they are applicable to few environments and place an additional burden of labeling every object with its security classification. In many practical situations, discretionary policies are preferred because they offer a better tradeoff between security and applicability than mandatory policies.

### 30.3.2 Role-Based Access Control

Role-based access control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprise-wide systems.

Its basic notion is that privileges and other permissions are associated with organizational **roles** rather than with individual users. Individual users are then assigned to appropriate roles. Roles can be created using the `CREATE ROLE` and `DESTROY ROLE` commands. The `GRANT` and `REVOKE` commands discussed in Section 30.2 can then be used to assign and revoke privileges from roles, as well as for individual users when needed. For example, a company may have roles such as sales account manager, purchasing agent, mailroom clerk, customer service manager, and so on. Multiple individuals can be assigned to each role. Security privileges that are common to a role are granted to the role name, and any individual assigned to this role would automatically have those privileges granted.

RBAC can be used with traditional discretionary and mandatory access controls; it ensures that only authorized users in their specified roles are given access to certain data or resources. Users create sessions during which they may activate a subset of roles to which they belong. Each session can be assigned to several roles, but it maps to one user or a single subject only. Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.

Separation of duties is another important requirement in various mainstream DBMSs. It is needed to prevent one user from doing work that requires the involvement of two or more people, thus preventing collusion. One method in which separation of duties can be successfully implemented is with mutual exclusion of roles. Two roles are said to be **mutually exclusive** if both the roles cannot be used simultaneously by the user. **Mutual exclusion of roles** can be categorized into two types, namely *authorization time exclusion (static)* and *runtime exclusion (dynamic)*. In authorization time exclusion, two roles that have been specified as mutually exclusive cannot be part of a user's authorization at the same time. In runtime exclusion, both these roles can be authorized to one user but cannot be activated by the user at the same time. Another variation in mutual exclusion of roles is that of complete and partial exclusion.

The **role hierarchy** in RBAC is a natural way to organize roles to reflect the organization's lines of authority and responsibility. By convention, junior roles at the bottom are connected to progressively senior roles as one moves up the hierarchy. The hierarchic diagrams are partial orders, so they are reflexive, transitive, and antisymmetric. In other words, if a user has one role, the user automatically has roles lower in the hierarchy. Defining a role hierarchy involves choosing the type of hierarchy and the roles, and then implementing the hierarchy by granting roles to other roles. Role hierarchy can be implemented in the following manner:

```
GRANT ROLE full_time TO employee_type1
GRANT ROLE intern TO employee_type2
```

The above are examples of granting the roles *full\_time* and *intern* to two types of employees.

Another issue related to security is *identity management*. **Identity** refers to a unique name of an individual person. Since the legal names of persons are not necessarily unique, the identity of a person must include sufficient additional information to

make the complete name unique. Authorizing this identity and managing the schema of these identities is called **identity management**. Identity management addresses how organizations can effectively authenticate people and manage their access to confidential information. It has become more visible as a business requirement across all industries affecting organizations of all sizes. Identity management administrators constantly need to satisfy application owners while keeping expenditures under control and increasing IT efficiency.

Another important consideration in RBAC systems is the possible temporal constraints that may exist on roles, such as the time and duration of role activations and the timed triggering of a role by an activation of another role. Using an RBAC model is a highly desirable goal for addressing the key security requirements of Web-based applications. Roles can be assigned to workflow tasks so that a user with any of the roles related to a task may be authorized to execute it and may play a certain role only for a certain duration.

RBAC models have several desirable features, such as flexibility, policy neutrality, better support for security management and administration, and a natural enforcement of the hierarchical organization structure within organizations. They also have other aspects that make them attractive candidates for developing secure Web-based applications. These features are lacking in DAC and MAC models. RBAC models do include the capabilities available in traditional DAC and MAC policies. Furthermore, an RBAC model provides mechanisms for addressing the security issues related to the execution of tasks and workflows, and for specifying user-defined and organization-specific policies. Easier deployment over the Internet has been another reason for the success of RBAC models.

### 30.3.3 Label-Based Security and Row-Level Access Control

Many mainstream RDBMSs currently use the concept of row-level access control, where sophisticated access control rules can be implemented by considering the data row by row. In row-level access control, each data row is given a label, which is used to store information about data sensitivity. Row-level access control provides finer granularity of data security by allowing the permissions to be set for each row and not just for the table or column. Initially the user is given a default session label by the database administrator. Levels correspond to a hierarchy of data-sensitivity levels to exposure or corruption, with the goal of maintaining privacy or security. Labels are used to prevent unauthorized users from viewing or altering certain data. A user having a low authorization level, usually represented by a low number, is denied access to data having a higher-level number. If no such label is given to a row, a row label is automatically assigned to it depending upon the user's session label.

A policy defined by an administrator is called a **label security policy**. Whenever data affected by the policy is accessed or queried through an application, the policy is automatically invoked. When a policy is implemented, a new column is added to each row in the schema. The added column contains the label for each row that

reflects the sensitivity of the row as per the policy. Similar to MAC (mandatory access control), where each user has a security clearance, each user has an identity in label-based security. This user's identity is compared to the label assigned to each row to determine whether the user has access to view the contents of that row. However, the user can write the label value himself, within certain restrictions and guidelines for that specific row. This label can be set to a value that is between the user's current session label and the user's minimum level. The DBA has the privilege to set an initial default row label.

The label security requirements are applied on top of the DAC requirements for each user. Hence, the user must satisfy the DAC requirements and then the label security requirements to access a row. The DAC requirements make sure that the user is legally authorized to carry on that operation on the schema. In most applications, only some of the tables need label-based security. For the majority of the application tables, the protection provided by DAC is sufficient.

Security policies are generally created by managers and human resources personnel. The policies are high-level, technology neutral, and relate to risks. Policies are a result of management instructions to specify organizational procedures, guiding principles, and courses of action that are considered to be expedient, prudent, or advantageous. Policies are typically accompanied by a definition of penalties and countermeasures if the policy is transgressed. These policies are then interpreted and converted to a set of label-oriented policies by the **label security administrator**, who defines the security labels for data and authorizations for users; these labels and authorizations govern access to specified protected objects.

Suppose a user has SELECT privileges on a table. When the user executes a SELECT statement on that table, label security will automatically evaluate each row returned by the query to determine whether the user has rights to view the data. For example, if the user has a sensitivity of 20, then the user can view all rows having a security level of 20 or lower. The level determines the sensitivity of the information contained in a row; the more sensitive the row, the higher its security label value. Such label security can be configured to perform security checks on UPDATE, DELETE, and INSERT statements as well.

### 30.3.4 XML Access Control

With the worldwide use of XML in commercial and scientific applications, efforts are under way to develop security standards. Among these efforts are digital signatures and encryption standards for XML. The XML Signature Syntax and Processing specification describes an XML syntax for representing the associations between cryptographic signatures and XML documents or other electronic resources. The specification also includes procedures for computing and verifying XML signatures. An XML digital signature differs from other protocols for message signing, such as **OpenPGP (Pretty Good Privacy)**—a confidentiality and authentication service that can be used for electronic mail and file storage application), in its support for signing only specific portions of the XML tree (see Chapter 13) rather than the



complete document. Additionally, the XML signature specification defines mechanisms for countersigning and transformations—so-called *canonicalization*—to ensure that two instances of the same text produce the same digest for signing even if their representations differ slightly; for example, in typographic white space.

The XML Encryption Syntax and Processing specification defines XML vocabulary and processing rules for protecting confidentiality of XML documents in whole or in part and of non-XML data as well. The encrypted content and additional processing information for the recipient are represented in well-formed XML so that the result can be further processed using XML tools. In contrast to other commonly used technologies for confidentiality, such as SSL (Secure Sockets Layer—a leading Internet security protocol) and virtual private networks, XML encryption also applies to parts of documents and to documents in persistent storage. Database systems such as PostgreSQL or Oracle support JSON (JavaScript Object Notation) objects as a data format and have similar facilities for JSON objects like those defined above for XML.

### 30.3.5 Access Control Policies for the Web and Mobile Applications

Publicly accessible Web application environments present a unique challenge to database security. These systems include those responsible for handling sensitive or private information and include social networks, mobile application API servers, and e-commerce transaction platforms.

Electronic commerce (**e-commerce**) environments are characterized by any transactions that are done electronically. They require elaborate access control policies that go beyond traditional DBMSs. In conventional database environments, access control is usually performed using a set of authorizations stated by security officers or users according to some security policies. Such a simple paradigm is not well suited for a dynamic environment like e-commerce. Furthermore, in an e-commerce environment the resources to be protected are not only traditional data but also knowledge and experience. Such peculiarities call for more flexibility in specifying access control policies. The access control mechanism must be flexible enough to support a wide spectrum of heterogeneous protection objects.

Because many reservation, ticketing, payment, and online shopping systems process information that is protected by law, the security architecture that goes beyond simple database access control must be put in place to protect the information. When an unauthorized party inappropriately accesses protected information, it amounts to a data breach, which has significant legal and financial consequences. This unauthorized party could be an adversary that actively seeks to steal protected information or may be an employee who overstepped his or her role or incorrectly distributed protected information to others. Inappropriate handling of credit card data, for instance, has led to significant data breaches at major retailers.

In conventional database environments, access control is usually performed using a set of authorizations stated by security officers. But in Web applications, it is all too



common that the Web application itself is the user rather than a duly authorized individual. This gives rise to a situation where the DBMS's access control mechanisms are bypassed and the database becomes just a relational data store to the system. In such environments, vulnerabilities like SQL injection (which we cover in depth in Section 30.4) become significantly more dangerous because it may lead to a total data breach rather than being limited to data that a particular account is authorized to access.

To protect against data breaches in these systems, a first requirement is a comprehensive information security policy that goes beyond the technical access control mechanisms found in mainstream DBMSs. Such a policy must protect not only traditional data, but also processes, knowledge, and experience.

A second related requirement is the support for content-based access control. **Content-based access control** allows one to express access control policies that take the protection object content into account. In order to support content-based access control, access control policies must allow inclusion of conditions based on the object content.

A third requirement is related to the heterogeneity of subjects, which requires access control policies based on user characteristics and qualifications rather than on specific and individual characteristics (for example, user IDs). A possible solution that will allow better accounting of user profiles in the formulation of access control policies, is to support the notion of credentials. A **credential** is a set of properties concerning a user that are relevant for security purposes (for example, age or position or role within an organization). For instance, by using credentials, one can simply formulate policies such as *Only permanent staff with five or more years of service can access documents related to the internals of the system.*

XML is expected to play a key role in access control for e-commerce applications<sup>6</sup> because XML is becoming the common representation language for document interchange over the Web, and is also becoming the language for e-commerce. Thus, on the one hand, there is the need to make XML representations secure by providing access control mechanisms specifically tailored to the protection of XML documents. On the other hand, access control information (that is, access control policies and user credentials) can be expressed using XML itself. The **Directory Services Markup Language (DSML)** is a representation of directory service information in XML syntax. It provides a foundation for a standard for communicating with the directory services that will be responsible for providing and authenticating user credentials. The uniform presentation of both protection objects and access control policies can be applied to policies and credentials themselves. For instance, some credential properties (such as the user name) may be accessible to everyone, whereas other properties may be visible only to a restricted class of users. Additionally, the use of an XML-based language for specifying credentials and access control policies facilitates secure credential submission and export of access control policies.

---

<sup>6</sup>See Thuraisingham et al. (2001).

## 30.4 SQL Injection

SQL injection is one of the most common threats to a database system. We will discuss it in detail later in this section. Some of the other frequent attacks on databases are:

- **Unauthorized privilege escalation.** This attack is characterized by an individual attempting to elevate his or her privilege by attacking vulnerable points in the database systems.
- **Privilege abuse.** Whereas unauthorized privilege escalation is done by an unauthorized user, this attack is performed by a privileged user. For example, an administrator who is allowed to change student information can use this privilege to update student grades without the instructor's permission.
- **Denial of service.** A **denial of service (DOS) attack** is an attempt to make resources unavailable to its intended users. It is a general attack category in which access to network applications or data is denied to intended users by overflowing the buffer or consuming resources.
- **Weak authentication.** If the user authentication scheme is weak, an attacker can impersonate the identity of a legitimate user by obtaining her login credentials.

### 30.4.1 SQL Injection Methods

As we discussed in Chapter 11, Web programs and applications that access a database can send commands and data to the database, as well as display data retrieved from the database through the Web browser. In an **SQL injection attack**, the attacker injects a string input through the application, which changes or manipulates the SQL statement to the attacker's advantage. An SQL injection attack can harm the database in various ways, such as unauthorized manipulation of the database or retrieval of sensitive data. It can also be used to execute system-level commands that may cause the system to deny service to the application. This section describes types of injection attacks.

**SQL Manipulation.** A manipulation attack, which is the most common type of injection attack, changes an SQL command in the application—for example, by adding conditions to the WHERE-clause of a query, or by expanding a query with additional query components using set operations such as UNION, INTERSECT, or MINUS. Other types of manipulation attacks are also possible. A typical manipulation attack occurs during database login. For example, suppose that a simplistic authentication procedure issues the following query and checks to see if any rows were returned:

```
SELECT * FROM users WHERE username = 'jake' and PASSWORD =
'jakespasswd' ;
```

The attacker can try to change (or manipulate) the SQL statement by changing it as follows:

```
SELECT * FROM users WHERE username = 'jake' and (PASSWORD =
'jakespasswd' or 'x' = 'x') ;
```

As a result, the attacker who knows that ‘jake’ is a valid login of some user is able to log into the database system as ‘jake’ without knowing his password and is able to do everything that ‘jake’ may be authorized to do to the database system.

**Code Injection.** This type of attack attempts to add additional SQL statements or commands to the existing SQL statement by exploiting a computer bug, which is caused by processing invalid data. The attacker can inject or introduce code into a computer program to change the course of execution. Code injection is a popular technique for system hacking or cracking to gain information.

**Function Call Injection.** In this kind of attack, a database function or operating system function call is inserted into a vulnerable SQL statement to manipulate the data or make a privileged system call. For example, it is possible to exploit a function that performs some aspect related to network communication. In addition, functions that are contained in a customized database package, or any custom database function, can be executed as part of an SQL query. In particular, dynamically created SQL queries (see Chapter 10) can be exploited since they are constructed at runtime.

For example, the *dual* table is used in the FROM clause of SQL in Oracle when a user needs to run SQL that does not logically have a table name. To get today’s date, we can use:

```
SELECT SYSDATE FROM dual;
```

The following example demonstrates that even the simplest SQL statements can be vulnerable.

```
SELECT TRANSLATE ('user input', 'from_string', 'to_string') FROM dual;
```

Here, TRANSLATE is used to replace a string of characters with another string of characters. The TRANSLATE function above will replace the characters of the ‘from\_string’ with the characters in the ‘to\_string’ one by one. This means that the *f* will be replaced with the *t*, the *r* with the *o*, the *o* with the *\_*, and so on.

This type of SQL statement can be subjected to a function injection attack. Consider the following example:

```
SELECT TRANSLATE (" || UTL_HTTP.REQUEST ('http://129.107.2.1/') || ",
'98765432', '9876') FROM dual;
```

The user can input the string (“ || UTL\_HTTP.REQUEST ('http://129.107.2.1/') ||”), where || is the concatenate operator, thus requesting a page from a Web server. UTL\_HTTP makes Hypertext Transfer Protocol (HTTP) callouts from SQL. The REQUEST object takes a URL ('http://129.107.2.1/' in this example) as a parameter, contacts that site, and returns the data (typically HTML) obtained from that site. The attacker could manipulate the string he inputs, as well as the URL, to include other functions and do other illegal operations. We just used a dummy example to show conversion of ‘98765432’ to ‘9876’, but the user’s intent would be to access the URL and get sensitive information. The attacker can then retrieve

useful information from the database server—located at the URL that is passed as a parameter—and send it to the Web server (that calls the TRANSLATE function).

### 30.4.2 Risks Associated with SQL Injection

SQL injection is harmful and the risks associated with it provide motivation for attackers. Some of the risks associated with SQL injection attacks are explained below.

- **Database fingerprinting.** The attacker can determine the type of database being used in the backend so that he can use database-specific attacks that correspond to weaknesses in a particular DBMS.
- **Denial of service.** The attacker can flood the server with requests, thus denying service to valid users, or the attacker can delete some data.
- **Bypassing authentication.** This is one of the most common risks, in which the attacker can gain access to the database as an authorized user and perform all the desired tasks.
- **Identifying injectable parameters.** In this type of attack, the attacker gathers important information about the type and structure of the back-end database of a Web application. This attack is made possible by the fact that the default error page returned by application servers is often overly descriptive.
- **Executing remote commands.** This provides attackers with a tool to execute arbitrary commands on the database. For example, a remote user can execute stored database procedures and functions from a remote SQL interactive interface.
- **Performing privilege escalation.** This type of attack takes advantage of logical flaws within the database to upgrade the access level.

### 30.4.3 Protection Techniques against SQL Injection

Protection against SQL injection attacks can be achieved by applying certain programming rules to all Web-accessible procedures and functions. This section describes some of these techniques.

**Bind Variables (Using Parameterized Statements).** The use of bind variables (also known as *parameters*; see Chapter 10) protects against injection attacks and also improves performance.

Consider the following example using Java and JDBC:

```
PreparedStatement stmt = conn.prepareStatement( "SELECT * FROM
EMPLOYEE WHERE EMPLOYEE_ID=? AND PASSWORD=?");
stmt.setString(1, employee_id);
stmt.setString(2, password);
```

Instead of embedding the user input into the statement, the input should be bound to a parameter. In this example, the input '1' is assigned (bound) to a bind variable

‘employee\_id’ and input ‘2’ to the bind variable ‘password’ instead of directly passing string parameters.

**Filtering Input (Input Validation).** This technique can be used to remove escape characters from input strings by using the SQL `ReplacE` function. For example, the delimiter single quote (‘) can be replaced by two single quotes (”). Some SQL manipulation attacks can be prevented by using this technique, since escape characters can be used to inject manipulation attacks. However, because there can be a large number of escape characters, this technique is not reliable.

**Function Security.** Database functions, both standard and custom, should be restricted, as they can be exploited in the SQL function injection attacks.

### 30.5 Introduction to Statistical Database Security

Statistical databases are used mainly to produce statistics about various populations. The database may contain confidential data about individuals; this information should be protected from user access. However, users are permitted to retrieve statistical information about the populations, such as averages, sums, counts, maximums, minimums, and standard deviations. The techniques that have been developed to protect the privacy of individual information are beyond the scope of this text. We will illustrate the problem with a very simple example, which refers to the relation shown in Figure 30.3. This is a `PERSON` relation with the attributes `Name`, `Ssn`, `Income`, `Address`, `City`, `State`, `Zip`, `Sex`, and `Last_degree`.

A **population** is a set of tuples of a relation (table) that satisfy some selection condition. Hence, each selection condition on the `PERSON` relation will specify a particular population of `PERSON` tuples. For example, the condition `Sex = ‘M’` specifies the male population; the condition `((Sex = ‘F’) AND (Last_degree = ‘M.S.’ OR Last_degree = ‘Ph.D.’))` specifies the female population that has an M.S. or Ph.D. degree as their highest degree; and the condition `City = ‘Houston’` specifies the population that lives in Houston.

Statistical queries involve applying statistical functions to a population of tuples. For example, we may want to retrieve the number of individuals in a population or the average income in the population. However, statistical users are not allowed to retrieve individual data, such as the income of a specific person. **Statistical database security** techniques must prohibit the retrieval of individual data. This can be achieved by prohibiting queries that retrieve attribute values and by allowing

**Figure 30.3**  
The `PERSON` relation schema for illustrating statistical database security.

PERSON								
Name	<u>Ssn</u>	Income	Address	City	State	Zip	Sex	Last_degree

only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION. Such queries are sometimes called **statistical queries**.

It is the responsibility of a database management system to ensure the confidentiality of information about individuals while still providing useful statistical summaries of data about those individuals to users. Provision of **privacy protection** of users in a statistical database is paramount; its violation is illustrated in the following example.

In some cases it is possible to **infer** the values of individual tuples from a sequence of statistical queries. This is particularly true when the conditions result in a population consisting of a small number of tuples. As an illustration, consider the following statistical queries:

**Q1: SELECT COUNT (\*) FROM PERSON**  
**WHERE** <condition>;

**Q2: SELECT AVG (Income) FROM PERSON**  
**WHERE** <condition>;

Now suppose that we are interested in finding the Salary of Jane Smith, and we know that she has a Ph.D. degree and that she lives in the city of Bellaire, Texas. We issue the statistical query Q1 with the following condition:

(Last\_degree='Ph.D.' AND Sex='F' AND City='Bellaire' AND State='Texas')

If we get a result of 1 for this query, we can issue Q2 with the same condition and find the Salary of Jane Smith. Even if the result of Q1 on the preceding condition is not 1 but is a small number—say 2 or 3—we can issue statistical queries using the functions MAX, MIN, and AVERAGE to identify the possible range of values for the Salary of Jane Smith.

The possibility of inferring individual information from statistical queries is reduced if no statistical queries are permitted whenever the number of tuples in the population specified by the selection condition falls below some threshold. Another technique for prohibiting retrieval of individual information is to prohibit sequences of queries that refer repeatedly to the same population of tuples. It is also possible to introduce slight inaccuracies or *noise* into the results of statistical queries deliberately, to make it difficult to deduce individual information from the results. Another technique is partitioning of the database. Partitioning implies that records are stored in groups of some minimum size; queries can refer to any complete group or set of groups, but never to subsets of records within a group. The interested reader is referred to the bibliography at the end of this chapter for a discussion of these techniques.

## 30.6 Introduction to Flow Control

**Flow control** regulates the distribution or flow of information among accessible objects. A flow between object *X* and object *Y* occurs when a program reads values from *X* and writes values into *Y*. **Flow controls** check that information contained in some objects does not flow explicitly or implicitly into less protected objects. Thus, a

user cannot get indirectly in  $Y$  what he or she cannot get directly in  $X$ . Active flow control began in the early 1970s. Most flow controls employ some concept of security class; the transfer of information from a sender to a receiver is allowed only if the receiver's security class is at least as privileged as the sender's. Examples of a flow control include preventing a service program from leaking a customer's confidential data, and blocking the transmission of secret military data to an unknown classified user.

A **flow policy** specifies the channels along which information is allowed to move. The simplest flow policy specifies just two classes of information—confidential ( $C$ ) and nonconfidential ( $N$ )—and allows all flows except those from class  $C$  to class  $N$ . This policy can solve the confinement problem that arises when a service program handles data such as customer information, some of which may be confidential. For example, an income-tax-computing service might be allowed to retain a customer's address and the bill for services rendered, but not a customer's income or deductions.

Access control mechanisms are responsible for checking users' authorizations for resource access: Only granted operations are executed. Flow controls can be enforced by an extended access control mechanism, which involves assigning a security class (usually called the *clearance*) to each running program. The program is allowed to read a particular memory segment only if its security class is as high as that of the segment. It is allowed to write in a segment only if its class is as low as that of the segment. This automatically ensures that no information transmitted by the person can move from a higher to a lower class. For example, a military program with a secret clearance can only read from objects that are unclassified and confidential and can only write into objects that are secret or top secret.

Two types of flow can be distinguished: *explicit flows*, which occur as a consequence of assignment instructions, such as  $Y := f(X_1, X_n)$ ; and *implicit flows*, which are generated by conditional instructions, such as if  $f(X_{m+1}, \dots, X_n)$  then  $Y := f(X_1, X_m)$ .

Flow control mechanisms must verify that only authorized flows, both explicit and implicit, are executed. A set of rules must be satisfied to ensure secure information flows. Rules can be expressed using flow relations among classes and assigned to information, stating the authorized flows within a system. (An information flow from  $A$  to  $B$  occurs when information associated with  $A$  affects the value of information associated with  $B$ . The flow results from operations that cause information transfer from one object to another.) These relations can define, for a class, the set of classes where information (classified in that class) can flow, or can state the specific relations to be verified between two classes to allow information to flow from one to the other. In general, flow control mechanisms implement the controls by assigning a label to each object and by specifying the security class of the object. Labels are then used to verify the flow relations defined in the model.

### 30.6.1 Covert Channels

A covert channel allows a transfer of information that violates the security or the policy. Specifically, a **covert channel** allows information to pass from a higher classification level to a lower classification level through improper means. Covert



channels can be classified into two broad categories: timing channels and storage. The distinguishing feature between the two is that in a **timing channel** the information is conveyed by the timing of events or processes, whereas **storage channels** do not require any temporal synchronization, in that information is conveyed by accessing system information or what is otherwise inaccessible to the user.

In a simple example of a covert channel, consider a distributed database system in which two nodes have user security levels of secret (S) and unclassified (U). In order for a transaction to commit, both nodes must agree to commit. They mutually can only do operations that are consistent with the \*-property, which states that in any transaction, the S site cannot write or pass information to the U site. However, if these two sites collude to set up a covert channel between them, a transaction involving secret data may be committed unconditionally by the U site, but the S site may do so in some predefined agreed-upon way so that certain information may be passed from the S site to the U site, violating the \*-property. This may be achieved where the transaction runs repeatedly, but the actions taken by the S site implicitly convey information to the U site. Measures such as locking, which we discussed in Chapters 21 and 22, prevent concurrent writing of the information by users with different security levels into the same objects, preventing the storage-type covert channels. Operating systems and distributed databases provide control over the multiprogramming of operations, which allows a sharing of resources without the possibility of encroachment of one program or process into another's memory or other resources in the system, thus preventing timing-oriented covert channels. In general, covert channels are not a major problem in well-implemented robust database implementations. However, certain schemes may be contrived by clever users that implicitly transfer information.

Some security experts believe that one way to avoid covert channels is to disallow programmers to actually gain access to sensitive data that a program will process after the program has been put into operation. For example, a programmer for a bank has no need to access the names or balances in depositors' accounts. Programmers for brokerage firms do not need to know what buy and sell orders exist for clients. During program testing, access to a form of real data or some sample test data may be justifiable, but not after the program has been accepted for regular use.

## 30.7 Encryption and Public Key Infrastructures

The previous methods of access and flow control, despite being strong control measures, may not be able to protect databases from some threats. Suppose we communicate data, but our data falls into the hands of a nonlegitimate user. In this situation, by using encryption we can disguise the message so that even if the transmission is diverted, the message will not be revealed. **Encryption** is the conversion of data into a form, called a **ciphertext**, that cannot be easily understood by unauthorized persons. It enhances security and privacy when access controls are bypassed, because in cases of data loss or theft, encrypted data cannot be easily understood by unauthorized persons.



With this background, we adhere to following standard definitions:<sup>7</sup>

- *Ciphertext*: Encrypted (enciphered) data
- *Plaintext (or cleartext)*: Intelligible data that has meaning and can be read or acted upon without the application of decryption
- *Encryption*: The process of transforming plaintext into ciphertext
- *Decryption*: The process of transforming ciphertext back into plaintext

Encryption consists of applying an **encryption algorithm** to data using some pre-specified **encryption key**. The resulting data must be **decrypted** using a **decryption key** to recover the original data.

### 30.7.1 The Data Encryption and Advanced Encryption Standards

The **Data Encryption Standard (DES)** is a system developed by the U.S. government for use by the general public. It has been widely accepted as a cryptographic standard both in the United States and abroad. DES can provide end-to-end encryption on the channel between sender *A* and receiver *B*. The DES algorithm is a careful and complex combination of two of the fundamental building blocks of encryption: substitution and permutation (transposition). The algorithm derives its strength from repeated application of these two techniques for a total of 16 cycles. Plaintext (the original form of the message) is encrypted as blocks of 64 bits. Although the key is 64 bits long, in effect the key can be any 56-bit number. After questioning the adequacy of DES, the NIST introduced the **Advanced Encryption Standard (AES)**. This algorithm has a block size of 128 bits, compared with DES's 56-bit block size, and can use keys of 128, 192, or 256 bits, compared with DES's 56-bit key. AES introduces more possible keys, compared with DES, and thus takes a much longer time to crack. In present systems, AES is the default with large key lengths. It is also the standard for full drive encryption products, with both Apple FileVault and Microsoft BitLocker using 256-bit or 128-bit keys. TripleDES is a fallback option if a legacy system cannot use a modern encryption standard.

### 30.7.2 Symmetric Key Algorithms

A symmetric key is one key that is used for both encryption and decryption. By using a symmetric key, fast encryption and decryption is possible for routine use with sensitive data in the database. A message encrypted with a secret key can be decrypted only with the same secret key. Algorithms used for symmetric key encryption are called **secret key algorithms**. Since secret-key algorithms are mostly used for encrypting the content of a message, they are also called **content-encryption algorithms**.

---

<sup>7</sup>U.S. Department of Commerce.

The major liability associated with secret-key algorithms is the need for sharing the secret key. A possible method is to derive the secret key from a user-supplied password string by applying the same function to the string at both the sender and receiver; this is known as a *password-based encryption algorithm*. The strength of the symmetric key encryption depends on the size of the key used. For the same algorithm, encrypting using a longer key is tougher to break than the one using a shorter key.

### 30.7.3 Public (Asymmetric) Key Encryption

In 1976, Diffie and Hellman proposed a new kind of cryptosystem, which they called **public key encryption**. Public key algorithms are based on mathematical functions rather than operations on bit patterns. They address one drawback of symmetric key encryption, namely that both sender and recipient must exchange the common key in a secure manner. In public key systems, two keys are used for encryption/decryption. The *public key* can be transmitted in a nonsecure way, whereas the *private key* is not transmitted at all. These algorithms—which use two related keys, a public key and a private key, to perform complementary operations (encryption and decryption)—are known as **asymmetric key encryption algorithms**. The use of two keys can have profound consequences in the areas of confidentiality, key distribution, and authentication. The two keys used for public key encryption are referred to as the **public key** and the **private key**. The private key is kept secret, but it is referred to as a *private key* rather than a *secret key* (the key used in conventional encryption) to avoid confusion with conventional encryption. The two keys are mathematically related, since one of the keys is used to perform encryption and the other to perform decryption. However, it is very difficult to derive the private key from the public key.

A public key encryption scheme, or *infrastructure*, has six ingredients:

1. **Plaintext.** This is the data or readable message that is fed into the algorithm as input.
2. **Encryption algorithm.** This algorithm performs various transformations on the plaintext.
3. and 4. **Public and private keys.** These are a pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exact transformations performed by the encryption algorithm depend on the public or private key that is provided as input. For example, if a message is encrypted using the public key, it can only be decrypted using the private key.
5. **Ciphertext.** This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different ciphertexts.
6. **Decryption algorithm.** This algorithm accepts the ciphertext and the matching key and produces the original plaintext.

As the name suggests, the public key of the pair is made public for others to use, whereas the private key is known only to its owner. A general-purpose public key

cryptographic algorithm relies on one key for encryption and a different but related key for decryption. The essential steps are as follows:

1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private.
3. If a sender wishes to send a private message to a receiver, the sender encrypts the message using the receiver's public key.
4. When the receiver receives the message, he or she decrypts it using the receiver's private key. No other recipient can decrypt the message because only the receiver knows his or her private key.

**The RSA Public Key Encryption Algorithm.** One of the first public key schemes was introduced in 1978 by Ron Rivest, Adi Shamir, and Len Adleman at MIT<sup>8</sup> and is named after them as the **RSA scheme**. The RSA scheme has since then reigned supreme as the most widely accepted and implemented approach to public key encryption. The RSA encryption algorithm incorporates results from number theory, combined with the difficulty of determining the prime factors of a target. The RSA algorithm also operates with modular arithmetic—mod  $n$ .

Two keys,  $d$  and  $e$ , are used for decryption and encryption. An important property is that they can be interchanged.  $n$  is chosen as a large integer that is a product of two large distinct prime numbers,  $a$  and  $b$ ,  $n = a \times b$ . The encryption key  $e$  is a randomly chosen number between 1 and  $n$  that is relatively prime to  $(a - 1) \times (b - 1)$ . The plaintext block  $P$  is encrypted as  $P^e$  where  $P^e = P \bmod n$ . Because the exponentiation is performed mod  $n$ , factoring  $P^e$  to uncover the encrypted plaintext is difficult. However, the decrypting key  $d$  is carefully chosen so that  $(P^e)d \bmod n = P$ . The decryption key  $d$  can be computed from the condition that  $d \times e = 1 \bmod ((a - 1) \times (b - 1))$ . Thus, the legitimate receiver who knows  $d$  simply computes  $(P^e)d \bmod n = P$  and recovers  $P$  without having to factor  $P^e$ .

### 30.7.4 Digital Signatures

A digital signature is an example of using encryption techniques to provide authentication services in electronic commerce applications. Like a handwritten signature, a **digital signature** is a means of associating a mark unique to an individual with a body of text. The mark should be unforgettable, meaning that others should be able to check that the signature comes from the originator.

A digital signature consists of a string of symbols. If a person's digital signature were always the same for each message, then one could easily counterfeit it by simply copying the string of symbols. Thus, signatures must be different for each use. This can be achieved by making each digital signature a function of the message

---

<sup>8</sup>Rivest et al. (1978).

that it is signing, together with a timestamp. To be unique to each signer and counterfeitproof, each digital signature must also depend on some secret number that is unique to the signer. Thus, in general, a counterfeitproof digital signature must depend on the message and a unique secret number of the signer. The verifier of the signature, however, should not need to know any secret number. Public key techniques are the best means of creating digital signatures with these properties.

### 30.7.5 Digital Certificates

A digital certificate is used to combine the value of a public key with the identity of the person or service that holds the corresponding private key into a digitally signed statement. Certificates are issued and signed by a certification authority (CA). The entity receiving this certificate from a CA is the subject of that certificate. Instead of requiring each participant in an application to authenticate every user, third-party authentication relies on the use of digital certificates.

The digital certificate itself contains various types of information. For example, both the certification authority and the certificate owner information are included. The following list describes all the information included in the certificate:

1. The certificate owner information, which is represented by a unique identifier known as the distinguished name (DN) of the owner. This includes the owner's name, as well as the owner's organization and other information about the owner.
2. The certificate also includes the public key of the owner.
3. The date of issue of the certificate is also included.
4. The validity period is specified by 'Valid From' and 'Valid To' dates, which are included in each certificate.
5. Issuer identifier information is included in the certificate.
6. Finally, the digital signature of the issuing CA for the certificate is included. All the information listed is encoded through a message-digest function, which creates the digital signature. The digital signature basically certifies that the association between the certificate owner and public key is valid.

## 30.8 Privacy Issues and Preservation

Preserving data privacy is a growing challenge for database security and privacy experts. In some perspectives, to preserve data privacy we should even limit performing large-scale data mining and analysis. The most commonly used techniques to address this concern are to avoid building mammoth central warehouses as a single repository of vital information. This is one of the stumbling blocks for creating nationwide registries of patients for many important diseases. Another possible measure is to intentionally modify or perturb data.

If all data were available at a single warehouse, violating only a single repository's security could expose all data. Avoiding central warehouses and using distributed

data mining algorithms minimizes the exchange of data needed to develop globally valid models. By modifying, perturbing, and anonymizing data, we can also mitigate privacy risks associated with data mining. This can be done by removing identity information from the released data and injecting noise into the data. However, by using these techniques, we should pay attention to the quality of the resulting data in the database, which may undergo too many modifications. We must be able to estimate the errors that may be introduced by these modifications.

Privacy is an important area of ongoing research in database management. It is complicated due to its multidisciplinary nature and the issues related to the subjectivity in the interpretation of privacy, trust, and so on. As an example, consider medical and legal records and transactions, which must maintain certain privacy requirements. Providing access control and privacy for mobile devices is also receiving increased attention. DBMSs need robust techniques for efficient storage of security-relevant information on small devices, as well as trust negotiation techniques. Where to keep information related to user identities, profiles, credentials, and permissions and how to use it for reliable user identification remains an important problem. Because large-sized streams of data are generated in such environments, efficient techniques for access control must be devised and integrated with processing techniques for continuous queries. Finally, the privacy of user location data, acquired from sensors and communication networks, must be ensured.

## **30.9 Challenges to Maintaining Database Security**

Considering the vast growth in volume and speed of threats to databases and information assets, research efforts need to be devoted to a number of issues: data quality, intellectual property rights, and database survivability, to name a few. We briefly outline the work required in a few important areas that researchers in database security are trying to address.

### **30.9.1 Data Quality**

The database community needs techniques and organizational solutions to assess and attest to the quality of data. These techniques may include simple mechanisms such as quality stamps that are posted on Web sites. We also need techniques that provide more effective integrity semantics verification and tools for the assessment of data quality, based on techniques such as record linkage. Application-level recovery techniques are also needed for automatically repairing incorrect data. The ETL (extract, transform, load) tools widely used to load data in data warehouses (see Section 29.4) are presently grappling with these issues.

### **30.9.2 Intellectual Property Rights**

With the widespread use of the Internet and intranets, legal and informational aspects of data are becoming major concerns for organizations. To address these

concerns, watermarking techniques for relational data have been proposed. The main purpose of digital watermarking is to protect content from unauthorized duplication and distribution by enabling provable ownership of the content. Digital watermarking has traditionally relied upon the availability of a large noise domain within which the object can be altered while retaining its essential properties. However, research is needed to assess the robustness of such techniques and to investigate different approaches aimed at preventing intellectual property rights violations.

### 30.9.3 Database Survivability

Database systems need to operate and continue their functions, even with reduced capabilities, despite disruptive events such as information warfare attacks. A DBMS, in addition to making every effort to prevent an attack and detecting one in the event of occurrence, should be able to do the following:

- **Confinement.** Take immediate action to eliminate the attacker's access to the system and to isolate or contain the problem to prevent further spread.
- **Damage assessment.** Determine the extent of the problem, including failed functions and corrupted data.
- **Reconfiguration.** Reconfigure to allow operation to continue in a degraded mode while recovery proceeds.
- **Repair.** Recover corrupted or lost data and repair or reinstall failed system functions to reestablish a normal level of operation.
- **Fault treatment.** To the extent possible, identify the weaknesses exploited in the attack and take steps to prevent a recurrence.

The goal of the information warfare attacker is to damage the organization's operation and fulfillment of its mission through disruption of its information systems. The specific target of an attack may be the system itself or its data. Although attacks that bring the system down outright are severe and dramatic, they must also be well timed to achieve the attacker's goal, since attacks will receive immediate and concentrated attention in order to bring the system back to operational condition, diagnose how the attack took place, and install preventive measures.

To date, issues related to database survivability have not been sufficiently investigated. Much more research needs to be devoted to techniques and methodologies that ensure database system survivability.

## 30.10 Oracle Label-Based Security

Restricting access to entire tables or isolating sensitive data into separate databases is a costly operation to administer. **Oracle label security** overcomes the need for such measures by enabling row-level access control. It is available starting with Oracle Database 11g Release 1 (11.1) Enterprise Edition. Each database table or view has a security policy associated with it. This policy executes every time the table or view is queried or altered. Developers can readily add label-based access

control to their Oracle Database applications. Label-based security provides an adaptable way of controlling access to sensitive data. Both users and data have labels associated with them. Oracle label security uses these labels to provide security.

### 30.10.1 Virtual Private Database (VPD) Technology

**Virtual private databases (VPDs)** are a feature of the Oracle Enterprise Edition that add predicates to user statements to limit their access in a transparent manner to the user and the application. The VPD concept allows server-enforced, fine-grained access control for a secure application.

VPD provides access control based on policies. These VPD policies enforce object-level access control or row-level security. VPD provides an application programming interface (API) that allows security policies to be attached to database tables or views. Using PL/SQL, a host programming language used in Oracle applications, developers and security administrators can implement security policies with the help of stored procedures.<sup>9</sup> VPD policies allow developers to remove access security mechanisms from applications and centralize them within the Oracle Database.

VPD is enabled by associating a security “policy” with a table, view, or synonym. An administrator uses the supplied PL/SQL package, DBMS\_RLS, to bind a policy function with a database object. When an object having a security policy associated with it is accessed, the function implementing this policy is consulted. The policy function returns a predicate (a WHERE clause) that is then appended to the user’s SQL statement, thus *transparently* and *dynamically* modifying the user’s data access. Oracle label security is a technique of enforcing row-level security in the form of a security policy.

### 30.10.2 Label Security Architecture

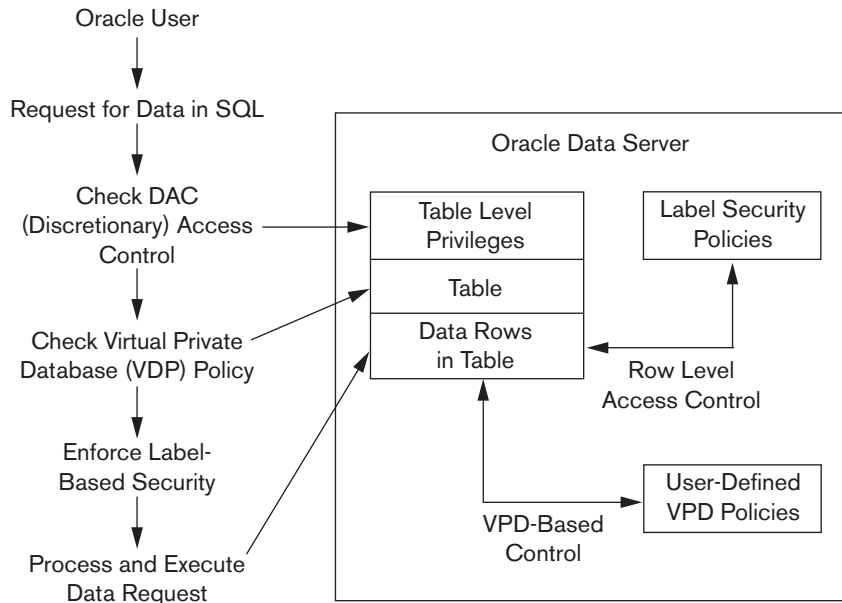
Oracle label security is built on the VPD technology delivered in the Oracle Database 11.1 Enterprise Edition. Figure 30.4 illustrates how data is accessed under Oracle label security, showing the sequence of DAC and label security checks.

Figure 30.4 shows the sequence of discretionary access control (DAC) and label security checks. The left part of the figure shows an application user in an Oracle Database 11g Release 1 (11.1) session sending out an SQL request. The Oracle DBMS checks the DAC privileges of the user, making sure that he or she has SELECT privileges on the table. Then it checks whether the table has a virtual private database (VPD) policy associated with it to determine if the table is protected using Oracle label security. If it is, the VPD SQL modification (WHERE clause) is added to the original SQL statement to find the set of accessible rows for the user to view. Then Oracle label security checks the labels on each row to determine the subset of rows to which the user has access (as explained in the next section). This modified query is processed, optimized, and executed.

---

<sup>9</sup>Stored procedures are discussed in Section 8.2.2.





**Figure 30.4**  
Oracle label security  
architecture.  
Data from: Oracle  
(2007)

### 30.10.3 How Data Labels and User Labels Work Together

A user's label indicates the information the user is permitted to access. It also determines the type of access (read or write) that the user has on that information. A row's label shows the sensitivity of the information that the row contains as well as the ownership of the information. When a table in the database has a label-based access associated with it, a row can be accessed only if the user's label meets certain criteria defined in the policy definitions. Access is granted or denied based on the result of comparing the data label and the session label of the user.

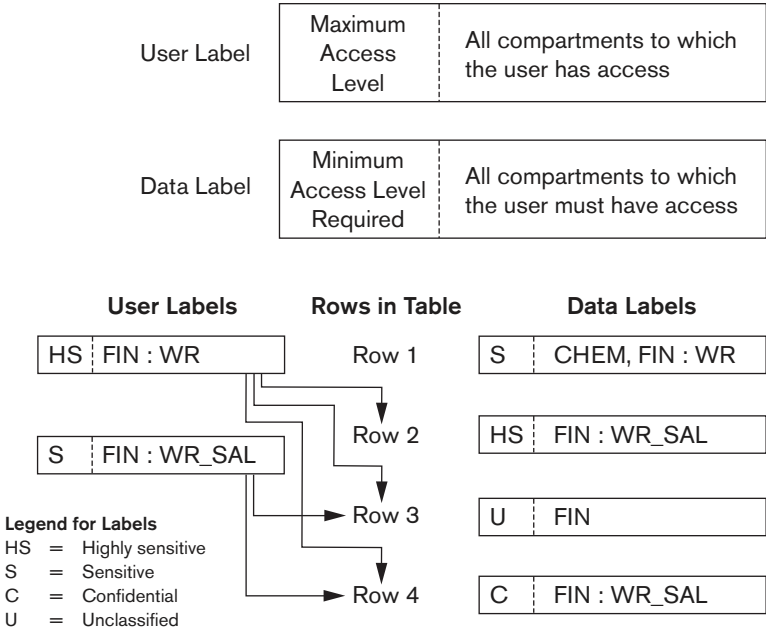
Compartments allow a finer classification of sensitivity of the labeled data. All data related to the same project can be labeled with the same compartment. Compartments are optional; a label can contain zero or more compartments.

Groups are used to identify organizations as owners of the data with corresponding group labels. Groups are hierarchical; for example, a group can be associated with a parent group.

If a user has a maximum level of SENSITIVE, then the user potentially has access to all data having levels SENSITIVE, CONFIDENTIAL, and UNCLASSIFIED. This user has no access to HIGHLY\_SENSITIVE data. Figure 30.5 shows how data labels and user labels work together to provide access control in Oracle label security.

As shown in Figure 30.5, User 1 can access the rows 2, 3, and 4 because his maximum level is HS (Highly\_Sensitive). He has access to the FIN (Finance) compartment, and his access to group WR (Western Region) hierarchically includes group





**Figure 30.5**  
Data labels and user labels  
in Oracle.  
Data from: Oracle (2007)

WR\_SAL (WR Sales). He cannot access row 1 because he does not have the CHEM (Chemical) compartment. It is important that a user has authorization for all compartments in a row's data label so the user can access that row. Based on this example, user 2 can access both rows 3 and 4 and has a maximum level of S, which is less than the HS in row 2. So, although user 2 has access to the FIN compartment, he can only access the group WR\_SAL and thus cannot access row 1.

### 30.11 Summary

In this chapter, we discussed several techniques for enforcing database system security. Section 30.1 is an introduction to database security. We presented in Section 30.1.1 different threats to databases in terms of loss of integrity, availability, and confidentiality. We discussed in Section 30.1.2 the types of control measures to deal with these problems: access control, inference control, flow control, and encryption. In the rest of Section 30.1, we covered various issues related to security, including data sensitivity and type of disclosures; security versus precision of results when a user requests information; and the relationship between information security and privacy.

Security enforcement deals with controlling access to the database system as a whole and controlling authorization to access specific portions of a database. The former is usually done by assigning accounts with passwords to users. The latter can be accomplished by using a system of granting and revoking privileges to individual accounts for accessing specific parts of the database. This approach,

presented in Section 30.2, is generally referred to as discretionary access control (DAC). We presented some SQL commands for granting and revoking privileges, and we illustrated their use with examples. Then in Section 30.3 we gave an overview of mandatory access control (MAC) mechanisms that enforce multilevel security. These require the classifications of users and data values into security classes and enforce the rules that prohibit flow of information from higher to lower security levels. Some of the key concepts underlying the multilevel relational model, including filtering and polyinstantiation, were presented. Role-based access control (RBAC) was introduced in Section 30.3.2, which assigns privileges based on roles that users play. We introduced the notion of role hierarchies, mutual exclusion of roles, and row- and label-based security. We explained the main ideas behind the threat of SQL injection in Section 30.4, the methods in which it can be induced, and the various types of risks associated with it. Then we gave an idea of the various ways SQL injection can be prevented.

We briefly discussed in Section 30.5 the problem of controlling access to statistical databases to protect the privacy of individual information while concurrently providing statistical access to populations of records. The issues related to flow control and the problems associated with covert channels were discussed next in Section 30.6, as well as encryption and public-versus-private key-based infrastructures in Section 30.7. The idea of symmetric key algorithms and the use of the popular asymmetric key-based public key infrastructure (PKI) scheme was explained in Section 30.7.3. We also covered in Sections 30.7.4 and 30.7.5 the concepts of digital signatures and digital certificates. We highlighted in Section 30.8 the importance of privacy issues and hinted at some privacy preservation techniques. We discussed in Section 30.9 a variety of challenges to security, including data quality, intellectual property rights, and data survivability. We ended the chapter in Section 30.10 by introducing the implementation of security policies by using a combination of label-based security and virtual private databases in Oracle 11g.

## Review Questions

- 30.1.** Discuss what is meant by each of the following terms: *database authorization, access control, data encryption, privileged (system) account, database audit, audit trail*.
- 30.2.** Which account is designated as the owner of a relation? What privileges does the owner of a relation have?
- 30.3.** How is the view mechanism used as an authorization mechanism?
- 30.4.** Discuss the types of privileges at the account level and those at the relation level.
- 30.5.** What is meant by granting a privilege? What is meant by revoking a privilege?
- 30.6.** Discuss the system of propagation of privileges and the restraints imposed by horizontal and vertical propagation limits.
- 30.7.** List the types of privileges available in SQL.

- 30.8. What is the difference between *discretionary* and *mandatory* access control?
- 30.9. What are the typical security classifications? Discuss the simple security property and the \*-property, and explain the justification behind these rules for enforcing multilevel security.
- 30.10. Describe the multilevel relational data model. Define the following terms: *apparent key*, *polyinstantiation*, *filtering*.
- 30.11. What are the relative merits of using DAC or MAC?
- 30.12. What is role-based access control? In what ways is it superior to DAC and MAC?
- 30.13. What are the two types of mutual exclusion in role-based access control?
- 30.14. What is meant by row-level access control?
- 30.15. What is label security? How does an administrator enforce it?
- 30.16. What are the different types of SQL injection attacks?
- 30.17. What risks are associated with SQL injection attacks?
- 30.18. What preventive measures are possible against SQL injection attacks?
- 30.19. What is a statistical database? Discuss the problem of statistical database security.
- 30.20. How is privacy related to statistical database security? What measures can be taken to ensure some degree of privacy in statistical databases?
- 30.21. What is flow control as a security measure? What types of flow control exist?
- 30.22. What are covert channels? Give an example of a covert channel.
- 30.23. What is the goal of encryption? What process is involved in encrypting data and then recovering it at the other end?
- 30.24. Give an example of an encryption algorithm and explain how it works.
- 30.25. Repeat the previous question for the popular RSA algorithm.
- 30.26. What is a symmetric key algorithm for key-based security?
- 30.27. What is the public key infrastructure scheme? How does it provide security?
- 30.28. What are digital signatures? How do they work?
- 30.29. What type of information does a digital certificate include?

## Exercises

- 30.30. How can privacy of data be preserved in a database?
- 30.31. What are some of the current outstanding challenges for database security?

- 30.32.** Consider the relational database schema in Figure 5.5. Suppose that all the relations were created by (and hence are owned by) user *X*, who wants to grant the following privileges to user accounts *A*, *B*, *C*, *D*, and *E*:
- Account *A* can retrieve or modify any relation except *DEPENDENT* and can grant any of these privileges to other users.
  - Account *B* can retrieve all the attributes of *EMPLOYEE* and *DEPARTMENT* except for *Salary*, *Mgr\_ssn*, and *Mgr\_start\_date*.
  - Account *C* can retrieve or modify *WORKS\_ON* but can only retrieve the *Fname*, *Minit*, *Lname*, and *Ssn* attributes of *EMPLOYEE* and the *Pname* and *Pnumber* attributes of *PROJECT*.
  - Account *D* can retrieve any attribute of *EMPLOYEE* or *DEPENDENT* and can modify *DEPENDENT*.
  - Account *E* can retrieve any attribute of *EMPLOYEE* but only for *EMPLOYEE* tuples that have *Dno* = 3.
  - Write SQL statements to grant these privileges. Use views where appropriate.
- 30.33.** Suppose that privilege (a) of Exercise 30.32 is to be given with *GRANT OPTION* but only so that account *A* can grant it to at most five accounts, and each of these accounts can propagate the privilege to other accounts but *without* the *GRANT OPTION* privilege. What would the horizontal and vertical propagation limits be in this case?
- 30.34.** Consider the relation shown in Figure 30.2(d). How would it appear to a user with classification *U*? Suppose that a classification *U* user tries to update the salary of ‘Smith’ to \$50,000; what would be the result of this action?

## Selected Bibliography

Authorization based on granting and revoking privileges was proposed for the SYSTEM R experimental DBMS and is presented in Griffiths and Wade (1976). Several books discuss security in databases and computer systems in general, including the books by Leiss (1982a), Fernandez et al. (1981), and Fugini et al. (1995). Natan (2005) is a practical book on security and auditing implementation issues in all major RDBMSs.

Many papers discuss different techniques for the design and protection of statistical databases. They include McLeish (1989), Chin and Ozsoyoglu (1981), Leiss (1982), Wong (1984), and Denning (1980). Ghosh (1984) discusses the use of statistical databases for quality control. There are also many papers discussing cryptography and data encryption, including Diffie and Hellman (1979), Rivest et al. (1978), Akl (1983), Pfleeger and Pfleeger (2007), Omura et al. (1990), Stallings (2000), and Iyer et al. (2004).

Halfond et al. (2006) helps us understand the concepts of SQL injection attacks and the various threats imposed by them. The white paper Oracle (2007a) explains how Oracle is less prone to SQL injection attack as compared to SQL Server. Oracle

(2007a) also gives a brief explanation of how these attacks can be prevented from occurring. Further proposed frameworks are discussed in Boyd and Keromytis (2004), Halfond and Orso (2005), and McClure and Krüger (2005).

Multilevel security is discussed in Jajodia and Sandhu (1991), Denning et al. (1987), Smith and Winslett (1992), Stachour and Thuraisingham (1990), Lunt et al. (1990), and Bertino et al. (2001). Overviews of research issues in database security are given by Lunt and Fernandez (1990), Jajodia and Sandhu (1991), Bertino (1998), Castano et al. (1995), and Thuraisingham et al. (2001). The effects of multilevel security on concurrency control are discussed in Atluri et al. (1997). Security in next-generation, semantic, and object-oriented databases is discussed in Rabbiti et al. (1991), Jajodia and Kogan (1990), and Smith (1990). Oh (1999) presents a model for both discretionary and mandatory security. Security models for Web-based applications and role-based access control are discussed in Joshi et al. (2001). Security issues for managers in the context of e-commerce applications and the need for risk assessment models for selection of appropriate security control measures are discussed in Farahmand et al. (2005). Row-level access control is explained in detail in Oracle (2007b) and Sybase (2005). The latter also provides details on role hierarchy and mutual exclusion. Oracle (2009) explains how Oracle uses the concept of identity management.

Recent advances as well as future challenges for security and privacy of databases are discussed in Bertino and Sandhu (2005). U.S. Govt. (1978), OECD (1980), and NRC (2003) are good references on the view of privacy by important government bodies. Karat et al. (2009) discusses a policy framework for security and privacy. XML and access control are discussed in Naedele (2003). More details are presented on privacy-preserving techniques in Vaidya and Clifton (2004), intellectual property rights in Sion et al. (2004), and database survivability in Jajodia et al. (1999). Oracle's VPD technology and label-based security is discussed in more detail in Oracle (2007b).

Agrawal et al. (2002) defined the concept of Hippocratic Databases for preserving privacy in healthcare information. K-anonymity as a privacy preserving technique is discussed in Bayardo and Agrawal (2005) and in Ciriani et al. (2007). Privacy-preserving data mining techniques based on k-anonymity are surveyed by Ciriani et al. (2008). Vimercati et al. (2014) discuss encryption and fragmentation as potential protection techniques for data confidentiality in the cloud.

## Alternative Diagrammatic Notations for ER Models

**F**igure A.1 shows a number of different diagrammatic notations for representing ER and EER model concepts. Unfortunately, there is no standard notation: different database design practitioners prefer different notations. Similarly, various **CASE** (computer-aided software engineering) tools and **OOA** (object-oriented analysis) methodologies use various notations. Some notations are associated with models that have additional concepts and constraints beyond those of the ER and EER models described in Chapters 7 through 9, while other models have fewer concepts and constraints. The notation we used in Chapter 7 is quite close to the original notation for ER diagrams, which is still widely used. We discuss some alternate notations here.

Figure A.1(a) shows different notations for displaying entity types/classes, attributes, and relationships. In Chapters 7 through 9, we used the symbols marked (i) in Figure A.1(a)—namely, rectangle, oval, and diamond. Notice that symbol (ii) for entity types/classes, symbol (ii) for attributes, and symbol (ii) for relationships are similar, but they are used by different methodologies to represent three different concepts. The straight line symbol (iii) for representing relationships is used by several tools and methodologies.

Figure A.1(b) shows some notations for attaching attributes to entity types. We used notation (i). Notation (ii) uses the third notation (iii) for attributes from Figure A.1(a). The last two notations in Figure A.1(b)—(iii) and (iv)—are popular in OOA methodologies and in some CASE tools. In particular, the last notation displays both the attributes and the methods of a class, separated by a horizontal line.

**Figure A.1**  
Alternative notations. (a) Symbols for entity type/class, attribute, and relationship. (b) Displaying attributes. (c) Displaying cardinality ratios. (d) Various (min, max) notations. (e) Notations for displaying specialization/generalization.

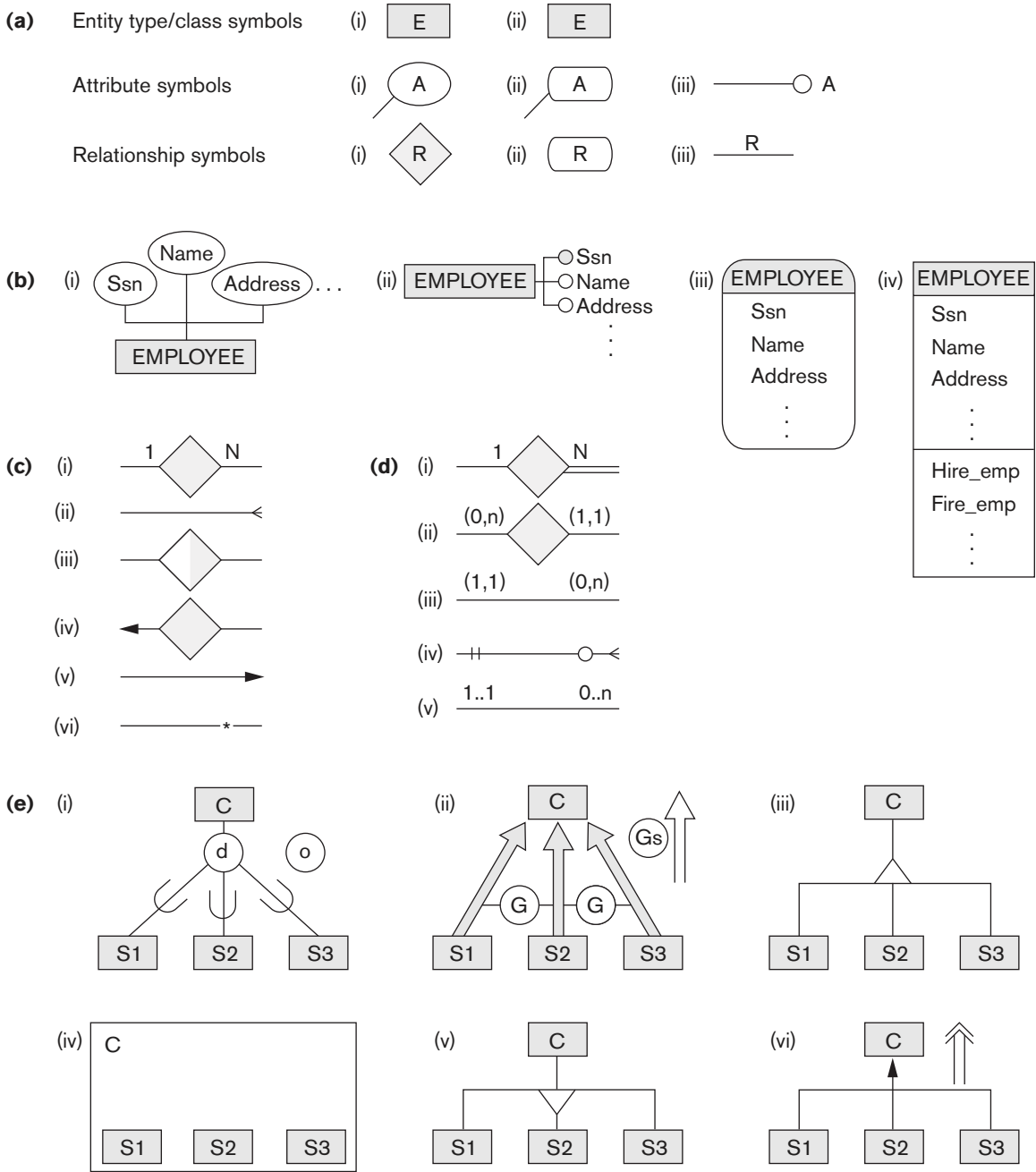


Figure A.1(c) shows various notations for representing the cardinality ratio of binary relationships. We used notation (i) in Chapters 7 through 9. Notation (ii)—known as the *chicken feet* notation—is quite popular. Notation (iv) uses the arrow as a functional reference (from the N to the 1 side) and resembles our notation for foreign keys in the relational model (see Figure 9.2); notation (v)—used in *Bachman diagrams* and the network data model—uses the arrow in the *reverse direction* (from the 1 to the N side). For a 1:1 relationship, (ii) uses a straight line without any chicken feet; (iii) makes both halves of the diamond white; and (iv) places arrowheads on both sides. For an M:N relationship, (ii) uses chicken feet at both ends of the line; (iii) makes both halves of the diamond black; and (iv) does not display any arrowheads.

Figure A.1(d) shows several variations for displaying (min, max) constraints, which are used to display both cardinality ratio and total/partial participation. We mostly used notation (i). Notation (ii) is the alternative notation we used in Figure 7.15 and discussed in Section 7.7.4. Recall that our notation specifies the constraint that each entity must participate in at least min and at most max relationship instances. Hence, for a 1:1 relationship, both max values are 1; for M:N, both max values are n. A min value greater than 0 (zero) specifies total participation (existence dependency). In methodologies that use the straight line for displaying relationships, it is common to *reverse the positioning* of the (min, max) constraints, as shown in (iii); a variation common in some tools (and in UML notation) is shown in (v). Another popular technique—which follows the same positioning as (iii)—is to display the *min* as o (“oh” or circle, which stands for zero) or as | (vertical dash, which stands for 1), and to display the max as | (vertical dash, which stands for 1) or as chicken feet (which stands for n), as shown in (iv).

Figure A.1(e) shows some notations for displaying specialization/generalization. We used notation (i) in Chapter 8, where a d in the circle specifies that the subclasses (S1, S2, and S3) are disjoint and an o in the circle specifies overlapping subclasses. Notation (ii) uses G (for generalization) to specify disjoint, and Gs to specify overlapping; some notations use the solid arrow, while others use the empty arrow (shown at the side). Notation (iii) uses a triangle pointing toward the superclass, and notation (v) uses a triangle pointing toward the subclasses; it is also possible to use both notations in the same methodology, with (iii) indicating generalization and (v) indicating specialization. Notation (iv) places the boxes representing subclasses within the box representing the superclass. Of the notations based on (vi), some use a single-lined arrow, and others use a double-lined arrow (shown at the side).

The notations shown in Figure A.1 show only some of the diagrammatic symbols that have been used or suggested for displaying database conceptual schemes. Other notations, as well as various combinations of the preceding, have also been used. It would be useful to establish a standard that everyone would adhere to, in order to prevent misunderstandings and reduce confusion.



This page intentionally left blank

## Parameters of Disks

The most important disk parameter is the time required to locate an arbitrary disk block, given its block address, and then to transfer the block between the disk and a main memory buffer. This is the random access time for accessing a disk block. There are three time components to consider as follows:

1. **Seek time (s).** This is the time needed to mechanically position the read/write head on the correct track for movable-head disks. (For fixed-head disks, it is the time needed to electronically switch to the appropriate read/write head.) For movable-head disks, this time varies, depending on the distance between the current track under the read/write head and the track specified in the block address. Usually, the disk manufacturer provides an average seek time in milliseconds. The typical range of average seek time is 4 to 10 msec. This is the main *culprit* for the delay involved in transferring blocks between disk and memory.
2. **Rotational delay (rd).** Once the read/write head is at the correct track, the user must wait for the beginning of the required block to rotate into position under the read/write head. On average, this takes about the time for half a revolution of the disk, but it actually ranges from immediate access (if the start of the required block is in position under the read/write head right after the seek) to a full disk revolution (if the start of the required block just passed the read/write head after the seek). If the speed of disk rotation is  $p$  revolutions per minute (rpm), then the average rotational delay  $rd$  is given by

$$rd = (1/2) * (1/p) \text{ min} = (60 * 1000)/(2 * p) \text{ msec} = 30000/p \text{ msec}$$

A typical value for  $p$  is 10,000 rpm, which gives a rotational delay of  $rd = 3$  msec. For fixed-head disks, where the seek time is negligible, this component causes the greatest delay in transferring a disk block.

3. **Block transfer time (*btt*)**. Once the read/write head is at the beginning of the required block, some time is needed to transfer the data in the block. This block transfer time depends on the block size, track size, and rotational speed. If the **transfer rate** for the disk is *tr* bytes/msec and the block size is *B* bytes, then

$$btt = B/tr \text{ msec}$$

If we have a track size of 50 Kbytes and *p* is 3600 rpm, then the transfer rate in bytes/msec is

$$tr = (50 * 1000)/(60 * 1000/3600) = 3000 \text{ bytes/msec}$$

In this case, *btt* = *B*/3000 msec, where *B* is the block size in bytes.

The average time (*s*) needed to find and transfer a block, given its block address, is estimated by

$$(s + rd + btt) \text{ msec}$$

This holds for either reading or writing a block. The principal method of reducing this time is to transfer several blocks that are stored on one or more tracks of the same cylinder; then the seek time is required for the first block only. To transfer consecutively *k noncontiguous* blocks that are on the same cylinder, we need approximately

$$s + (k * (rd + btt)) \text{ msec}$$

In this case, we need two or more buffers in main storage because we are continuously reading or writing the *k* blocks, as we discussed in Chapter 17. The transfer time per block is reduced even further when *consecutive blocks* on the same track or cylinder are transferred. This eliminates the rotational delay for all but the first block, so the estimate for transferring *k* consecutive blocks is

$$s + rd + (k * btt) \text{ msec}$$

A more accurate estimate for transferring consecutive blocks takes into account the interblock gap (see Section 17.2.1), which includes the information that enables the read/write head to determine which block it is about to read. Usually, the disk manufacturer provides a **bulk transfer rate (*btr*)** that takes the gap size into account when reading consecutively stored blocks. If the gap size is *G* bytes, then

$$btr = (B/(B + G)) * tr \text{ bytes/msec}$$

The bulk transfer rate is the rate of transferring *useful bytes* in the data blocks. The disk read/write head must go over all bytes on a track as the disk rotates, including the bytes in the interblock gaps, which store control information but not real data. When the bulk transfer rate is used, the time needed to transfer the useful data in one block out of several consecutive blocks is *B/btr*. Hence, the estimated time to read *k* blocks consecutively stored on the same cylinder becomes

$$s + rd + (k * (B/btr)) \text{ msec}$$

Another parameter of disks is the **rewrite time**. This is useful in cases when we read a block from the disk into a main memory buffer, update the buffer, and then write the buffer back to the same disk block on which it was stored. In many cases, the time required to update the buffer in main memory is less than the time required for one disk revolution. If we know that the buffer is ready for rewriting, the system can keep the disk heads on the same track, and during the next disk revolution the updated buffer is rewritten back to the disk block. Hence, the rewrite time  $T_{rw}$ , is usually estimated to be the time needed for one disk revolution:

$$T_{rw} = 2 * rd \text{ msec} = 60000/p \text{ msec}$$

To summarize, the following is a list of the parameters we have discussed and the symbols we use for them:

Seek time:	$s$ msec
Rotational delay:	$rd$ msec
Block transfer time:	$btt$ msec
Rewrite time:	$T_{rw}$ msec
Transfer rate:	$tr$ bytes/msec
Bulk transfer rate:	$btr$ bytes/msec
Block size:	$B$ bytes
Interblock gap size:	$G$ bytes
Disk speed:	$p$ rpm (revolutions per minute)

This page intentionally left blank

## Overview of the QBE Language

The Query-By-Example (QBE) language is important because it is one of the first graphical query languages with minimum syntax developed for database systems. It was developed at IBM Research and is available as an IBM commercial product as part of the QMF (Query Management Facility) interface option to DB2. The language was also implemented in the Paradox DBMS, and is related to a point-and-click type interface in the Microsoft Access DBMS. It differs from SQL in that the user does not have to explicitly specify a query using a fixed syntax; rather, the query is formulated by filling in **templates** of relations that are displayed on a monitor screen. Figure C.1 shows how these templates may look for the database of Figure 3.5. The user does not have to remember the names of attributes or relations because they are displayed as part of these templates. Additionally, the user does not have to follow rigid syntax rules for query specification; rather, constants and variables are entered in the columns of the templates to construct an **example** related to the retrieval or update request. QBE is related to the domain relational calculus, as we shall see, and its original specification has been shown to be relationally complete.

### C.1 Basic Retrievals in QBE

In QBE retrieval queries are specified by filling in one or more rows in the templates of the tables. For a single relation query, we enter either constants or **example elements** (a QBE term) in the columns of the template of that relation. An example element stands for a domain variable and is specified as an example value preceded by the underscore character ( ). Additionally, a P. prefix (called the P dot operator) is entered in certain columns to indicate that we would like to print (or display)

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT\_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS\_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

**Figure C.1**  
The relational schema of Figure 3.5 as it may be displayed by QBE.

values in those columns for our result. The constants specify values that must be exactly matched in those columns.

For example, consider the query Q0: *Retrieve the birth date and address of John B. Smith.* In Figures C.2(a) through C.2(d) we show how this query can be specified in a progressively more terse form in QBE. In Figure C.2(a) an example of an employee is presented as the type of row that we are interested in. By leaving John B. Smith as constants in the Fname, Minit, and Lname columns, we are specifying an exact match in those columns. The rest of the columns are preceded by an underscore indicating that they are domain variables (example elements). The P. prefix is placed in the Bdate and Address columns to indicate that we would like to output value(s) in those columns.

Q0 can be abbreviated as shown in Figure C.2(b). There is no need to specify example values for columns in which we are not interested. Moreover, because example values are completely arbitrary, we can just specify variable names for them, as shown in Figure C.2(c). Finally, we can also leave out the example values entirely, as shown in Figure C.2(d), and just specify a P. under the columns to be retrieved.

To see how retrieval queries in QBE are similar to the domain relational calculus, compare Figure C.2(d) with Q0 (simplified) in domain calculus as follows:

$$Q0 : \{ uv \mid EMPLOYEE(qrstuvwxyz) \text{ and } q='John' \text{ and } r='B' \text{ and } s='Smith'\}$$

## (a) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	_123456789	P_9/1/60	P_100 Main, Houston, TX	_M	_25000	_123456789	_3

## (b) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith		P_9/1/60	P_100 Main, Houston, TX				

## (c) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith		P_X	P_Y				

## (d) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith		P.	P.				

**Figure C.2**

Four ways to specify the query Q0 in QBE.

We can think of each column in a QBE template as an *implicit domain variable*; hence, Fname corresponds to the domain variable  $q$ , Minit corresponds to  $r$ , ..., and Dno corresponds to  $z$ . In the QBE query, the columns with P. correspond to variables specified to the left of the bar in domain calculus, whereas the columns with constant values correspond to tuple variables with equality selection conditions on them. The condition EMPLOYEE(qrstuvwxyz) and the existential quantifiers are implicit in the QBE query because the template corresponding to the EMPLOYEE relation is used.

In QBE, the user interface first allows the user to choose the tables (relations) needed to formulate a query by displaying a list of all relation names. Then the templates for the chosen relations are displayed. The user moves to the appropriate columns in the templates and specifies the query. Special function keys are provided to move among templates and perform certain functions.

We now give examples to illustrate basic facilities of QBE. Comparison operators other than = (such as > or  $\geq$ ) may be entered in a column before typing a constant value. For example, the query Q0A: *List the social security numbers of employees who work more than 20 hours per week on project number 1* can be specified as shown in Figure C.3(a). For more complex conditions, the user can ask for a **condition box**, which is created by pressing a particular function key. The user can then type the complex condition.<sup>1</sup>

<sup>1</sup>Negation with the  $\neg$  symbol is not allowed in a condition box.



**Figure C.3**

Specifying complex conditions in QBE. (a) The query Q0A. (b) The query Q0B with a condition box. (c) The query Q0B without a condition box.

(a)

Essn	Pno	Hours
P.		> 20

**WORKS\_ON**

(b)

Essn	Pno	Hours
P.	_PX	_HX

**CONDITIONS**

_HX > 20 and (PX = 1 or PX = 2)
---------------------------------

**WORKS\_ON**

(c)

Essn	Pno	Hours
P.	1	> 20
P.	2	> 20

For example, the query Q0B: *List the social security numbers of employees who work more than 20 hours per week on either project 1 or project 2* can be specified as shown in Figure C.3(b).

Some complex conditions can be specified without a condition box. The rule is that all conditions specified on the same row of a relation template are connected by the **and** logical connective (*all* must be satisfied by a selected tuple), whereas conditions specified on distinct rows are connected by **or** (*at least one* must be satisfied). Hence, Q0B can also be specified, as shown in Figure C.3(c), by entering two distinct rows in the template.

Now consider query Q0C: *List the social security numbers of employees who work on both project 1 and project 2*; this cannot be specified as in Figure C.4(a), which lists those who work on *either* project 1 or project 2. The example variable \_ES will bind itself to Essn values in <-, 1, -> tuples *as well as* to those in <-, 2, -> tuples. Figure C.4(b) shows how to specify Q0C correctly, where the condition (\_EX = \_EY) in the box makes the \_EX and \_EY variables bind only to identical Essn values.

In general, once a query is specified, the resulting values are displayed in the template under the appropriate columns. If the result contains more rows than can be displayed on the screen, most QBE implementations have function keys to allow scrolling up and down the rows. Similarly, if a template or several templates are too wide to appear on the screen, it is possible to scroll sideways to examine all the templates.

A join operation is specified in QBE by using the *same variable*<sup>2</sup> in the columns to be joined. For example, the query Q1: *List the name and address of all employees who*

<sup>2</sup>A variable is called an **example element** in QBE manuals.

<b>WORKS_ON</b>			
(a)	Essn	Pno	Hours
	P._ES	1	
	P._ES	2	

<b>WORKS_ON</b>			
(b)	Essn	Pno	Hours
	P._EX	1	
	P._EY	2	

<b>CONDITIONS</b>			
	_EX = _EY		

**Figure C.4**

Specifying EMPLOYEES who work on both projects. (a) Incorrect specification of an AND condition. (b) Correct specification.

work for the 'Research' department can be specified as shown in Figure C.5(a). Any number of joins can be specified in a single query. We can also specify a **result table** to display the result of the join query, as shown in Figure C.5(a); this is needed if the result includes attributes from two or more relations. If no result table is specified, the system provides the query result in the columns of the various relations, which may make it difficult to interpret. Figure C.5(a) also illustrates the feature of QBE for specifying that all attributes of a relation should be retrieved, by placing the P. operator under the relation name in the relation template.

To join a table with itself, we specify different variables to represent the different references to the table. For example, query Q8: *For each employee retrieve the employee's first and last name as well as the first and last name of his or her immediate supervisor* can be specified as shown in Figure C.5(b), where the variables starting with E refer to an employee and those starting with S refer to a supervisor.

## C.2 Grouping, Aggregation, and Database Modification in QBE

Next, consider the types of queries that require grouping or aggregate functions. A grouping operator *G.* can be specified in a column to indicate that tuples should be grouped by the value of that column. Common functions can be specified, such as AVG., SUM., CNT. (count), MAX., and MIN. In QBE the functions AVG., SUM., and CNT. are applied to distinct values within a group in the default case. If we want these functions to apply to all values, we must use the prefix ALL.<sup>3</sup> This convention is *different* in SQL, where the default is to apply a function to all values.

<sup>3</sup>ALL in QBE is unrelated to the universal quantifier.

## (a) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
_FN		_LN			_Addr				_DX

## DEPARTMENT

Dname	Dnumber	Mgrssn	Mgr_start_date
Research	_DX		

RESULT			
P.	_FN	_LN	_Addr

## (b) EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
_E1		_E2						_Xssn	
_S1		_S2	_Xssn						

RESULT				
P.	_E1	_E2	_S1	_S2

**Figure C.5**

Illustrating JOIN and result relations in QBE. (a) The query Q1. (b) The query Q8.

Figure C.6(a) shows query Q23, which counts the number of *distinct* salary values in the EMPLOYEE relation. Query Q23A (Figure C.6(b)) counts all salary values, which is the same as counting the number of employees. Figure C.6(c) shows Q24, which retrieves each department number and the number of employees and average salary within each department; hence, the Dno column is used for grouping as indicated by the G. function. Several of the operators G., P., and ALL can be specified in a single column. Figure C.6(d) shows query Q26, which displays each project name and the number of employees working on it for projects on which more than two employees work.

QBE has a negation symbol,  $\neg$ , which is used in a manner similar to the NOT EXISTS function in SQL. Figure C.7 shows query Q6, which lists the names of employees who have no dependents. The negation symbol  $\neg$  says that we will select values of the \_SX variable from the EMPLOYEE relation only if they do not occur in the DEPENDENT relation. The same effect can be produced by placing a  $\neg$  \_SX in the Essn column.

Although the QBE language as originally proposed was shown to support the equivalent of the EXISTS and NOT EXISTS functions of SQL, the QBE implementation in QMF (under the DB2 system) does *not* provide this support. Hence, the QMF version of QBE, which we discuss here, is *not relationally complete*. Queries such as Q3: *Find employees who work on all projects controlled by department 5* cannot be specified.

**(a) EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
							P.CNT.		

**(b) EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
							P.CNT.ALL		

**(c) EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
			P.CNT.ALL				P.AVG.ALL		P.G.

**(d) PROJECT**

Pname	Pnumber	Plocation	Dnum
P.	_PX		

**WORKS\_ON**

Essn	Pno	Hours
P.CNT.EX	G._PX	

**CONDITIONS**

CNT._EX > 2
-------------

**Figure C.6**

Functions and grouping in QBE.

(a) The query Q23. (b) The query Q23A.

(c) The query Q24. (d) The query Q26.

**EMPLOYEE**

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
P.		P.	_SX						

**DEPENDENT**

Essn	Dependent_name	Sex	Bdate	Relationship
_SX				

**Figure C.7**

Illustrating negation by the query Q6.

There are three QBE operators for modifying the database: I. for insert, D. for delete, and U. for update. The insert and delete operators are specified in the template column under the relation name, whereas the update operator is specified under the columns to be updated. Figure C.8(a) shows how to insert a new EMPLOYEE tuple. For deletion, we first enter the D. operator and then specify the tuples to be deleted by a condition (Figure C.8(b)). To update a tuple, we specify the U. operator under the attribute name, followed by the new value of the attribute. We should also select the tuple or tuples to be updated in the usual way. Figure C.8(c) shows an update

## (a) EMPLOYEE

	Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
I.	Richard	K	Marini	653298653	30-Dec-52	98 Oak Forest, Katy, TX	M	37000	987654321	4

## (b) EMPLOYEE

	Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
D.				653298653						

## (c) EMPLOYEE

	Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
	John		Smith					U_S*1.1		U.4

**Figure C.8**

Modifying the database in QBE. (a) Insertion. (b) Deletion. (c) Update in QBE.

request to increase the salary of 'John Smith' by 10 percent and also to reassign him to department number 4.

QBE also has data definition capabilities. The tables of a database can be specified interactively, and a table definition can also be updated by adding, renaming, or removing a column. We can also specify various characteristics for each column, such as whether it is a key of the relation, what its data type is, and whether an index should be created on that field. QBE also has facilities for view definition, authorization, storing query definitions for later use, and so on.

QBE does not use the *linear* style of SQL; rather, it is a *two-dimensional* language because users specify a query moving around the full area of the screen. Tests on users have shown that QBE is easier to learn than SQL, especially for nonspecialists. In this sense, QBE was the *first* user-friendly *visual* relational database language.

More recently, numerous other user-friendly interfaces have been developed for commercial database systems. The use of menus, graphics, and forms is now becoming quite common. Filling forms partially to issue a search request is akin to using QBE. Visual query languages, which are still not so common, are likely to be offered with commercial relational databases in the future.

# Bibliography

## Abbreviations Used in the Bibliography

ACM: Association for Computing Machinery  
AFIPS: American Federation of Information Processing Societies  
ASPLOS: Proceedings of the international Conference on Architectural Support for Programming Languages and Operating Systems  
CACM: Communications of the ACM (journal)  
CIKM: Proceedings of the International Conference on Information and Knowledge Management  
DASFAA: Proceedings of the International Conference on Database Systems for Advanced Applications  
DKE: Data and Knowledge Engineering, Elsevier Publishing (journal)  
EDBT: Proceedings of the International Conference on Extending Database Technology  
EDS: Proceedings of the International Conference on Expert Database Systems  
ER Conference: Proceedings of the International Conference on Entity-Relationship Approach (now called International Conference on Conceptual Modeling)  
ICDCS: Proceedings of the IEEE International Conference on Distributed Computing Systems  
ICDE: Proceedings of the IEEE International Conference on Data Engineering  
IEEE: Institute of Electrical and Electronics Engineers  
IEEE Computer: Computer magazine (journal) of the IEEE CS  
IEEE CS: IEEE Computer Society  
IFIP: International Federation for Information Processing  
JACM: Journal of the ACM  
KDD: Knowledge Discovery in Databases  
LNCS: Lecture Notes in Computer Science  
NCC: Proceedings of the National Computer Conference (published by AFIPS)  
OOPSLA: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications  
OSDI: USENIX Symposium on Operating Systems Design and Implementation  
PAMI: Pattern Analysis and Machine Intelligence  
PODS: Proceedings of the ACM Symposium on Principles of Database Systems

SIGMETRICS: Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems  
SIGMOD: Proceedings of the ACM SIGMOD International Conference on Management of Data  
SOSP: ACM Symposium on Operating System Principles  
TKDE: IEEE Transactions on Knowledge and Data Engineering (journal)  
TOCS: ACM Transactions on Computer Systems (journal)  
TODS: ACM Transactions on Database Systems (journal)  
TOIS: ACM Transactions on Information Systems (journal)  
TOOIS: ACM Transactions on Office Information Systems (journal)  
TPDS: IEEE Transactions of Parallel and Distributed Systems (journal)  
TSE: IEEE Transactions on Software Engineering (journal)  
VLDB: Proceedings of the International Conference on Very Large Data Bases (issues after 1981 available from Morgan Kaufmann, Menlo Park, California)

## Format for Bibliographic Citations

Book titles are in boldface—for example, **Database Computers**. Conference proceedings names are in italics—for example, *ACM Pacific Conference*. Journal names are in boldface—for example, **TODS** or **Information Systems**. For journal citations, we give the volume number and issue number (within the volume, if any) and date of issue. For example, “**TODS**, 3:4, December 1978” refers to the December 1978 issue of *ACM Transactions on Database Systems*, which is Volume 3, Number 4. Articles that appear in books or conference proceedings that are themselves cited in the bibliography are referenced as “in” these references—for example, “in *VLDB* [1978]” or “in Rustin [1974].” Page numbers (abbreviated “pp.”) are provided with pp. at the end of the citation whenever available. For citations with more than four authors, we will give the first author only followed by et al. In the selected bibliography at the end of each chapter, we use et al. if there are more than two authors.

## Bibliographic References

- Abadi, D. J., Madden, S. R., and Hachem, N. [2008] "Column Stores vs. Row Stores: How Different Are They Really?" in *SIGMOD* [2008].
- Abbott, R., and Garcia-Molina, H. [1989] "Scheduling Real-Time Transactions with Disk Resident Data," in *VLDB* [1989].
- Abiteboul, S., and Kanellakis, P. [1989] "Object Identity as a Query Language Primitive," in *SIGMOD* [1989].
- Abiteboul, S., Hull, R., and Vianu, V. [1995] **Foundations of Databases**, Addison-Wesley, 1995.
- Abramova, V. and Bernardino, J. [2013] "NoSQL Databases: MongoDB vs Cassandra," Proc. Sixth Int. Conf. on Comp. Sci. and Software Engg. (C<sup>3</sup>S<sup>2</sup>E'13), Porto, Portugal, July 2013, pp. 14–22.
- Abrial, J. [1974] "Data Semantics," in Klimbie and Koffeman [1974].
- Acharya, S., Alonso, R., Franklin, M., and Zdonik, S. [1995] "Broadcast Disks: Data Management for Asymmetric Communication Environments," in *SIGMOD* [1995].
- Adam, N., and Gongopadhyay, A. [1993] "Integrating Functional and Data Modeling in a Computer Integrated Manufacturing System," in *ICDE* [1993].
- Adriaans, P., and Zantinge, D. [1996] **Data Mining**, Addison-Wesley, 1996.
- Afsarmanesh, H., McLeod, D., Knapp, D., and Parker, A. [1985] "An Extensible Object-Oriented Approach to Databases for VLSI/CAD," in *VLDB* [1985].
- Afrati, F. and Ullman, J. [2010] "Optimizing Joins in a MapReduce Environment," in *EDBT* [2010].
- Agneeswaran, V.S. [2014] **Big Data Analytics Beyond Hadoop: Real-Time Applications with Storm, Spark, and More Hadoop Alternatives**, Pearson FT Press, 2014, 240 pp.
- Agrawal, D., and ElAbbadi, A. [1990] "Storage Efficient Replicated Databases," *TKDE*, 2:3, September 1990.
- Agrawal, R. et al. [2008] "**The Claremont Report on Database Research**," available at <http://db.cs.berkeley.edu/claremont/claremontreport08.pdf>, May 2008.
- Agrawal, R., and Gehani, N. [1989] "ODE: The Language and the Data Model," in *SIGMOD* [1989].
- Agrawal, R., and Srikant, R. [1994] "Fast Algorithms for Mining Association Rules in Large Databases," in *VLDB* [1994].
- Agrawal, R., Gehani, N., and Srinivasan, J. [1990] "OdeView: The Graphical Interface to Ode," in *SIGMOD* [1990].
- Agrawal, R., Imielinski, T., and Swami, A. [1993] "Mining Association Rules Between Sets of Items in Databases," in *SIGMOD* [1993].
- Agrawal, R., Imielinski, T., and Swami, A. [1993b] "Database Mining: A Performance Perspective," *TKDE* 5:6, December 1993.
- Agrawal, R., Mehta, M., Shafer, J., and Srikant, R. [1996] "The Quest Data Mining System," in *KDD* [1996].
- Ahad, R., and Basu, A. [1991] "ESQL: A Query Language for the Relational Model Supporting Image Domains," in *ICDE* [1991].
- Ahmed R. et al. [2006] "Cost-Based Query Transformation in Oracle," in *VLDB* [2006].
- Ahmed R. et al. [2014] "Of Snowstorms and Bushy Trees," in *VLDB* [2014].
- Aho, A., and Ullman, J. [1979] "Universality of Data Retrieval Languages," *Proc. POPL Conference*, San Antonio, TX, ACM, 1979.
- Aho, A., Beeri, C., and Ullman, J. [1979] "The Theory of Joins in Relational Databases," *TODS*, 4:3, September 1979.
- Aho, A., Sagiv, Y., and Ullman, J. [1979a] "Efficient Optimization of a Class of Relational Expressions," *TODS*, 4:4, December 1979.
- Akl, S. [1983] "Digital Signatures: A Tutorial Survey," *IEEE Computer*, 16:2, February 1983.
- Alagic, S. [1999] "A Family of the ODMG Object Models," in *Advances in Databases and Information Systems, Third East European Conference, ADBIS'99*, Maribor, Slovenia, J. Eder, I. Rozman, T. Welzer (eds.), September 1999, LNCS, No. 1691, Springer.
- Alashqur, A., Su, S., and Lam, H. [1989] "OQL: A Query Language for Manipulating Object-Oriented Databases," in *VLDB* [1989].
- Albano, A., Cardelli, L., and Orsini, R. [1985] "GALILEO: A Strongly Typed Interactive Conceptual Language," *TODS*, 10:2, June 1985, pp. 230–260.
- Albrecht J. H., [1996] "Universal GIS Operations," University of Osnabrueck, Germany, Ph.D. Dissertation, 1996.
- Allen, F., Loomis, M., and Mannino, M. [1982] "The Integrated Dictionary/Directory System," *ACM Computing Surveys*, 14:2, June 1982.
- Allen, J. [1983] "Maintaining Knowledge about Temporal Intervals," in *CACM* 26:11, November 1983, pp. 832–843.
- Alonso, G., Agrawal, D., El Abbadi, A., and Mohan, C. [1997] "Functionalities and Limitations of Current Workflow Management Systems," *IEEE Expert*, 1997.
- Amir, A., Feldman, R., and Kashi, R. [1997] "A New and Versatile Method for Association Generation," *Information Systems*, 22:6, September 1997.
- Ananthanarayanan, G. et al. [2012] "PACMan: Coordinated Memory Caching for Parallel Jobs," In *USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, 2012.
- Anderson, S. et al. [1981] "Sequence and Organization of the Human Mitochondrial Genome," *Nature*, 290: 457–465, 1981.



- Andrews, T., and Harris, C. [1987] "Combining Language and Database Advances in an Object-Oriented Development Environment," *OOPSLA*, 1987.
- ANSI [1975] American National Standards Institute Study Group on Data Base Management Systems: Interim Report, FDT, 7:2, ACM, 1975.
- ANSI [1986] American National Standards Institute: **The Database Language SQL**, Document ANSI X3.135, 1986.
- ANSI [1986a] American National Standards Institute: **The Database Language NDL**, Document ANSI X3.133, 1986.
- ANSI [1989] American National Standards Institute: **Information Resource Dictionary Systems**, Document ANSI X3.138, 1989.
- Antenucci, J. et al. [1998] **Geographic Information Systems: A Guide to the Technology**, Chapman and Hall, May 1998.
- Anwar, T., Beck, H., and Navathe, S. [1992] "Knowledge Mining by Imprecise Querying: A Classification Based Approach," in *ICDE* [1992].
- Apers, P., Hevner, A., and Yao, S. [1983] "Optimization Algorithms for Distributed Queries," *TSE*, 9:1, January 1983.
- Apweiler, R., Martin, M., O'Donovan, C., and Prues, M. [2003] "Managing Core Resources for Genomics and Proteomics," *Pharmacogenomics*, 4:3, May 2003, pp. 343–350.
- Aref, W. et al. [2004] "VDBMS: A Testbed Facility or Research in Video Database Benchmarking," in **Multi-media Systems (MMS)**, 9:6, June 2004, pp. 98–115.
- Arisawa, H., and Catarci, T. [2000] Advances in Visual Information Management, Proc. Fifth Working Conf. On Visual Database Systems, Arisawa, H., Catarci, T. (eds.), Fujitsu, Japan, *IFIP Conference Proceedings* 168, Kluwer, 2000.
- Armstrong, W. [1974] "Dependency Structures of Data Base Relationships," *Proc. IFIP Congress*, 1974.
- Ashburner, M. et al. [2000] "Gene Ontology: Tool for the unification of biology," *Nature Genetics*, Vol. 25, May 2000, pp. 25–29.
- Astrahan, M. et al. [1976] "System R: A Relational Approach to Data Base Management," *TODS*, 1:2, June 1976.
- Atkinson, M., and Buneman, P. [1987] "Types and Persistence in Database Programming Languages" in **ACM Computing Surveys**, 19:2, June 1987.
- Atkinson, Malcolm et al. [1990] The Object-Oriented Database System Manifesto, *Proc. Deductive and Object Oriented Database Conf. (DOOD)*, Kyoto, Japan, 1990.
- Atluri, V. et al. [1997] "Multilevel Secure Transaction Processing: Status and Prospects," in **Database Security: Status and Prospects**, Chapman and Hall, 1997, pp. 79–98.
- Atzeni, P., and De Antonellis, V. [1993] **Relational Database Theory**, Benjamin/Cummings, 1993.
- Atzeni, P., Mecca, G., and Merialdo, P. [1997] "To Weave the Web," in *VLDB* [1997].
- Bachman, C. [1969] "Data Structure Diagrams," **Data Base** (Bulletin of ACM SIGFIDET), 1:2, March 1969.
- Bachman, C. [1973] "The Programmer as a Navigator," *CACM*, 16:1, November 1973.
- Bachman, C. [1974] "The Data Structure Set Model," in Rustin [1974].
- Bachman, C., and Williams, S. [1964] "A General Purpose Programming System for Random Access Memories," *Proc. Fall Joint Computer Conference*, AFIPS, 26, 1964.
- Badal, D., and Popek, G. [1979] "Cost and Performance Analysis of Semantic Integrity Validation Methods," in *SIGMOD* [1979].
- Badrinath, B., and Imielinski, T. [1992] "Replication and Mobility," *Proc. Workshop on the Management of Replicated Data 1992*: pp. 9–12
- Badrinath, B., and Ramamritham, K. [1992] "Semantics-Based Concurrency Control: Beyond Commutativity," *TODS*, 17:1, March 1992.
- Bahga, A. and Madiseti, V. [2013] **Cloud Computing—A Hands On Approach**, (www.cloudcomputingbook.info), 2013, 454 pp.
- Baeza-Yates, R., and Larson, P. A. [1989] "Performance of B<sup>+</sup>-trees with Partial Expansions," *TKDE*, 1:2, June 1989.
- Baeza-Yates, R., and Ribero-Neto, B. [1999] **Modern Information Retrieval**, Addison-Wesley, 1999.
- Balbin, I., and Ramamohanrao, K. [1987] "A Generalization of the Different Approach to Recursive Query Evaluation," *Journal of Logic Programming*, 15:4, 1987.
- Bancilhon, F. [1985] "Naive Evaluation of Recursively Defined Relations," in **On Knowledge Base Management Systems** (Brodie, M., and Mylopoulos, J., eds.), Islamorada workshop 1985, Springer, pp. 165–178.
- Bancilhon, F., and Buneman, P., eds. [1990] **Advances in Database Programming Languages**, ACM Press, 1990.
- Bancilhon, F., and Ferran, G. [1995] "The ODMG Standard for Object Databases," *DASFAA 1995*, Singapore, pp. 273–283.
- Bancilhon, F., and Ramakrishnan, R. [1986] "An Amateur's Introduction to Recursive Query Processing Strategies," in *SIGMOD* [1986].
- Bancilhon, F., Delobel, C., and Kanellakis, P., eds. [1992] **Building an Object-Oriented Database System: The Story of O2**, Morgan Kaufmann, 1992.
- Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. [1986] "Magic Sets and Other Strange Ways to Implement Logic Programs," *PODS* [1986].
- Banerjee, J. et al. [1987] "Data Model Issues for Object-Oriented Applications," *TOOIS*, 5:1, January 1987.



- Banerjee, J., Kim, W., Kim, H., and Korth, H. [1987a] "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *SIGMOD* [1987].
- Barbara, D. [1999] "Mobile Computing and Databases – A Survey," *TKDE*, 11:1, January 1999.
- Baroody, A., and DeWitt, D. [1981] "An Object-Oriented Approach to Database System Implementation," *TODS*, 6:4, December 1981.
- Barrett T. et al. [2005] "NCBI GEO: mining millions of expression profiles—database and tools," *Nucleic Acid Research*, 33: database issue, 2005, pp. 562–566.
- Barrett, T. et al. [2007] "NCBI GEO: mining tens of millions of expression profiles—database and tools update," in *Nucleic Acids Research*, 35:1, January 2007.
- Barsalou, T., Siambela, N., Keller, A., and Wiederhold, G. [1991] "Updating Relational Databases Through Object-Based Views," in *SIGMOD* [1991].
- Bassiouni, M. [1988] "Single-Site and Distributed Optimistic Protocols for Concurrency Control," *TSE*, 14:8, August 1988.
- Batini, C., Ceri, S., and Navathe, S. [1992] *Database Design: An Entity-Relationship Approach*, Benjamin/Cummings, 1992.
- Batini, C., Lenzerini, M., and Navathe, S. [1987] "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys*, 18:4, December 1987.
- Batory, D. et al. [1988] "GENESIS: An Extensible Database Management System," *TSE*, 14:11, November 1988.
- Batory, D., and Buchmann, A. [1984] "Molecular Objects, Abstract Data Types, and Data Models: A Framework," in *VLDB* [1984].
- Bay, H., Tuytelaars, T., and Gool, L. V. [2006] "SURF: Speeded Up Robust Features," in *Proc. Ninth European Conference on Computer Vision*, May 2006.
- Bayer, R., and McCreight, E. [1972] "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, 1:3, February 1972.
- Bayer, R., Graham, M., and Seegmuller, G., eds. [1978] *Operating Systems: An Advanced Course*, Springer-Verlag, 1978.
- Beck, H., Anwar, T., and Navathe, S. [1994] "A Conceptual Clustering Algorithm for Database Schema Design," *TKDE*, 6:3, June 1994.
- Beck, H., Gala, S., and Navathe, S. [1989] "Classification as a Query Processing Technique in the CANDIDE Semantic Data Model," in *ICDE* [1989].
- Beeri, C., and Ramakrishnan, R. [1987] "On the Power of Magic" in *PODS* [1987].
- Beeri, C., Fagin, R., and Howard, J. [1977] "A Complete Axiomatization for Functional and Multivalued Dependencies," in *SIGMOD* [1977].
- Bellamkonda, S., et al., [2009], "Enhanced Subquery Optimization in Oracle", in *VLDB* [2009]
- Bell, D.E., and L. J. Lapadula, L.J. [1976]. **Secure computer system: Unified exposition and Multics Interpretation**, Technical Report MTR-2997, MITRE Corp., Bedford, MA, March 1976.
- Ben-Zvi, J. [1982] "The Time Relational Model," Ph.D. dissertation, University of California, Los Angeles, 1982.
- Benson, D., Boguski, M., Lipman, D., and Ostell, J., "GenBank," *Nucleic Acids Research*, 24:1, 1996.
- Benson, D., Karsch-Mizrachi, I., Lipman, D. et al. [2002] "GenBank," *Nucleic Acids Research*, 30:1, January 2008.
- Berg, B., and Roth, J. [1989] **Software for Optical Storage**, Meckler, 1989.
- Bergman, M. K. [2001] "The Deep Web: Surfacing Hidden Value," *The Journal of Electronic Publishing*, 7:1, August 2001.
- Berners-Lee, T., Caillian, R., Grooff, J., Pollermann, B. [1992] "World-Wide Web: The Information Universe," **Electronic Networking: Research, Applications and Policy**, 1:2, 1992.
- Berners-Lee, T., Caillian, R., Lautonen, A., Nielsen, H., and Secret, A. [1994] "The World Wide Web," *CACM*, 13:2, August 1994.
- Bernstein, P. [1976] "Synthesizing Third Normal Form Relations from Functional Dependencies," *TODS*, 1:4, December 1976.
- Bernstein, P. and Goodman, N. [1983] "Multiversion Concurrency Control—Theory and Algorithms," *TODS*, 8:4, pp. 465–483.
- Bernstein, P., and Goodman, N. [1980] "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," in *VLDB* [1980].
- Bernstein, P., and Goodman, N. [1981a] "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, 13:2, June 1981.
- Bernstein, P., and Goodman, N. [1981b] "The Power of Natural Semijoins," *SIAM Journal of Computing*, 10:4, December 1981.
- Bernstein, P., and Goodman, N. [1984] "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases," *TODS*, 9:4, December 1984.
- Bernstein, P., Blaustein, B., and Clarke, E. [1980] "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data," in *VLDB* [1980].
- Bernstein, P., Hadzilacos, V., and Goodman, N. [1987] *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
- Bertino, E. [1992] "Data Hiding and Security in Object-Oriented Databases," in *ICDE* [1992].
- Bertino, E. [1998] "Data Security," in *DKE* 25:1–2, pp. 199–216.

- Bertino, E. and Sandhu, R., [2005] "Security—Concepts, Approaches, and Challenges," in **IEEE Transactions on Dependable Secure Computing (TDSC)**, 2:1, 2005, pp. 2–19.
- Bertino, E., and Guerrini, G. [1998] "Extending the ODMG Object Model with Composite Objects," *OOPSLA*, Vancouver, Canada, 1998, pp. 259–270.
- Bertino, E., and Kim, W. [1989] "Indexing Techniques for Queries on Nested Objects," **TKDE**, 1:2, June 1989.
- Bertino, E., Catania, B., and Ferrari, E. [2001] "A Nested Transaction Model for Multilevel Secure Database Management Systems," **ACM Transactions on Information and System Security (TISSEC)**, 4:4, November 2001, pp. 321–370.
- Bertino, E., Negri, M., Pelagatti, G., and Sbatella, L. [1992] "Object-Oriented Query Languages: The Notion and the Issues," **TKDE**, 4:3, June 1992.
- Bertino, E., Pagani, E., and Rossi, G. [1992] "Fault Tolerance and Recovery in Mobile Computing Systems," in Kumar and Han [1992].
- Bertino, F., Rabitti, F., and Gibbs, S. [1988] "Query Processing in a Multimedia Document System," **TOIS**, 6:1, 1988.
- Bhargava, B., and Helal, A. [1993] "Efficient Reliability Mechanisms in Distributed Database Systems," *CIKM*, November 1993.
- Bhargava, B., and Reidl, J. [1988] "A Model for Adaptable Systems for Transaction Processing," in *ICDE* [1988].
- Bikel, D. and Zitouni, I. [2012] **Multilingual Natural Language Processing Applications: From Theory to Practice**, IBM Press, 2012.
- Biliris, A. [1992] "The Performance of Three Database Storage Structures for Managing Large Objects," in *SIGMOD* [1992].
- Biller, H. [1979] "On the Equivalence of Data Base Schemas—A Semantic Approach to Data Translation," **Information Systems**, 4:1, 1979.
- Bischoff, J., and T. Alexander, eds., **Data Warehouse: Practical Advice from the Experts**, Prentice-Hall, 1997.
- Biskup, J., Dayal, U., and Bernstein, P. [1979] "Synthesizing Independent Database Schemas," in *SIGMOD* [1979].
- Bitton, D., and Gray, J. [1988] "Disk Shadowing," in *VLDB* [1988], pp. 331–338.
- Bjork, A. [1973] "Recovery Scenario for a DB/DC System," *Proc. ACM National Conference*, 1973.
- Bjorner, D., and Lovengren, H. [1982] "Formalization of Database Systems and a Formal Definition of IMS," in *VLDB* [1982].
- Blaha, M., and Rumbaugh, J. [2005] **Object-Oriented Modeling and Design with UML**, 2nd ed., Prentice-Hall, 2005.
- Blaha, M., and Premerlani, W. [1998] **Object-Oriented Modeling and Design for Database Applications**, Prentice-Hall, 1998.
- Blakely, J., Larson, P. and Tompa, F.W. [1986] "Efficiently Updating Materialized Views," in *SIGMOD* [1986], pp. 61–71.
- Blakeley, J., and Martin, N. [1990] "Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis," in *ICDE* [1990].
- Blakeley, J., Coburn, N., and Larson, P. [1989] "Updated Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," **TODS**, 14:3, September 1989.
- Blasgen, M. et al. [1981] "System R: An Architectural Overview," **IBM Systems Journal**, 20:1, January 1981.
- Blasgen, M., and Eswaran, K. [1976] "On the Evaluation of Queries in a Relational Database System," **IBM Systems Journal**, 16:1, January 1976.
- Blei, D.M., Ng, A.Y., and Jordan, M.I. [2003] "Latent Dirichlet Allocation," *Journal of Machine. Learning. Research*, 3, March 2003, pp. 993–1022.
- Bleier, R., and Vorhaus, A. [1968] "File Organization in the SDC TDMS," *Proc. IFIP Congress*.
- Bocca, J. [1986] "EDUCE—A Marriage of Convenience: Prolog and a Relational DBMS," *Proc. Third International Conference on Logic Programming*, Springer-Verlag, 1986.
- Bocca, J. [1986a] "On the Evaluation Strategy of EDUCE," in *SIGMOD* [1986].
- Bodorick, P., Riordon, J., and Pyra, J. [1992] "Deciding on Correct Distributed Query Processing," **TKDE**, 4:3, June 1992.
- Boncz, P., Zukowski, M., and Nes, N. [2005] "MonetDB/X100: Hyper-Pipelining Query Execution," in *Proc. Conf. on Innovative Data Systems Research CIDR* [2005].
- Bonnet, P., Gehrke, J., and Seshadri, P. [2001] "Towards Sensor Database Systems," in *Proc. 2nd Int. Conf. on Mobile Data Management*, Hong Kong, China, LNCS 1987, Springer, January 2001, pp. 3–14.
- Booch, G., Rumbaugh, J., and Jacobson, I., **Unified Modeling Language User Guide**, Addison-Wesley, 1999.
- Borges, K., Laender, A., and Davis, C. [1999] "Spatial data integrity constraints in object oriented geographic data modeling," *Proc. 7th ACM International Symposium on Advances in Geographic Information Systems*, 1999.
- Borgida, A., Brachman, R., McGuinness, D., and Resnick, L. [1989] "CLASSIC: A Structural Data Model for Objects," in *SIGMOD* [1989].
- Borkin, S. [1978] "Data Model Equivalence," in *VLDB* [1978].
- Bossomaier, T., and Green, D. [2002] **Online GIS and Metadata**, Taylor and Francis, 2002.

- Boukerche, A., and Tuck, T. [2001] "Improving Concurrency Control in Distributed Databases with Predeclared Tables," in *Proc. Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference*, Manchester, UK August 28–31, 2001, pp. 301–309.
- Boutselakis, H. et al. [2003] "E-MSD: the European Bioinformatics Institute Macromolecular Structure Database," *Nucleic Acids Research*, 31:1, January 2003, pp. 458–462.
- Bouzeghoub, M., and Metais, E. [1991] "Semantic Modeling of Object-Oriented Databases," in *VLDB* [1991].
- Boyce, R., Chamberlin, D., King, W., and Hammer, M. [1975] "Specifying Queries as Relational Expressions," *CACM*, 18:11, November 1975.
- Boyd, S., and Keromytis, A. [2004] "SQLrand: Preventing SQL injection attacks," in *Proc. 2nd Applied Cryptography and Network Security Conf. (ACNS 2004)*, June 2004, pp. 292–302.
- Braam, P., and Schwan, P. [2002] Lustre: The intergalactic file system, *Proc. Ottawa Linux Symposium*, June 2002. (<http://ols.fedoraproject.org/OLS/Reprints-2002/braam-reprint.pdf>)
- Bracchi, G., Paolini, P., and Pelagatti, G. [1976] "Binary Logical Associations in Data Modelling," in Nijssen [1976].
- Brachman, R., and Levesque, H. [1984] "What Makes a Knowledge Base Knowledgeable? A View of Databases from the Knowledge Level," in *EDS* [1984].
- Brandon, M. et al. [2005] MITOMAP: A human mitochondrial genome database—2004 Update, *Nucleic Acid Research*, 34:1, January 2005.
- Bratbergengen, K. [1984] "Hashing Methods and Relational Algebra Operators," in *VLDB* [1984].
- Bray, O. [1988] **Computer Integrated Manufacturing—The Data Management Strategy**, Digital Press, 1988.
- Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S., Silberschatz, A. [1999] "Update Propagation Protocols for Replicated Databases," in *SIGMOD* [1999], pp. 97–108.
- Breitbart, Y., Silberschatz, A., and Thompson, G. [1990] "Reliable Transaction Management in a Multidatabase System," in *SIGMOD* [1990].
- Brinkhoff, T., Kriegel, H.-P., and Seeger, B. [1993] "Efficient Processing of Spatial Joins Using R-trees," in *SIGMOD* [1993].
- Broder, A. [2002] "A Taxonomy of Web Search," in **SIGIR Forum**, 36:2, September 2002, pp.3–10
- Brodeur, J., Bédard, Y., and Proulx, M. [2000] "Modelling Geospatial Application Databases Using UML-Based Repositories Aligned with International Standards in Geomatics," *Proc. 8th ACM International Symposium on Advances in Geographic Information Systems*. Washington, DC, ACM Press, 2000, pp. 39–46.
- Brodie, M., and Mylopoulos, J., eds. [1985] **On Knowledge Base Management Systems**, Springer-Verlag, 1985.
- Brodie, M., Mylopoulos, J., and Schmidt, J., eds. [1984] **On Conceptual Modeling**, Springer-Verlag, 1984.
- Brose, M., and Shneiderman, B. [1978] "Two Experimental Comparisons of Relational and Hierarchical Database Models," **International Journal of Man-Machine Studies**, 1978.
- Bruno, N., Chaudhuri, S., and Gravano, L. [2002] "Top-k Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation," **ACM TODS**, 27:2, 2002, pp. 153–187.
- Bry, F. [1990] "Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled," **DKE**, 5, 1990, pp. 289–312.
- Buckley, C., Salton, G., and Allan, J. [1993] "The SMART Information Retrieval Project," In *Proc. of the Workshop on Human Language Technology*, Human Language Technology Conference, Association for Computational Linguistics, March 1993.
- Bukhres, O. [1992] "Performance Comparison of Distributed Deadlock Detection Algorithms," in *ICDE* [1992].
- Buneman, P., and Frankel, R. [1979] "FQL: A Functional Query Language," in *SIGMOD* [1979].
- Burkhard, W. [1976] "Hashing and Trie Algorithms for Partial Match Retrieval," **TODS**, 1:2, June 1976, pp. 175–187.
- Burkhard, W. [1979] "Partial-match Hash Coding: Benefits of Redundancy," **TODS**, 4:2, June 1979, pp. 228–239.
- Bush, V. [1945] "As We May Think," *Atlantic Monthly*, 176:1, January 1945. Reprinted in Kochen, M., ed., **The Growth of Knowledge**, Wiley, 1967.
- Butterworth, P. Otis, A., and Stein, J. [1991] : "The Gemstone Object Database Management System," in **CACM**, 34:10, October 1991, pp. 64–77.
- Byte [1995] Special Issue on Mobile Computing, June 1995.
- CACM [1995] Special issue of the **Communications of the ACM**, on Digital Libraries, 38:5, May 1995.
- CACM [1998] Special issue of the **Communications of the ACM** on Digital Libraries: Global Scope and Unlimited Access, 41:4, April 1998.
- Cahill, M.J., Rohm, U., and Fekete, A. [2008] "Serializable Isolation for Snapshot Databases," in *SIGMOD* [2008].
- Cammarata, S., Ramachandra, P., and Shane, D. [1989] "Extending a Relational Database with Deferred Referential Integrity Checking and Intelligent Joins," in *SIGMOD* [1989].
- Campbell, D., Embley, D., and Czejdo, B. [1985] "A Relationally Complete Query Language for the Entity-Relationship Model," in *ER Conference* [1985].
- Cardenas, A. [1985] **Data Base Management Systems**, 2nd ed., Allyn and Bacon, 1985.

- Carey, M. et al. [1986] "The Architecture of the EXODUS Extensible DBMS," in Dittrich and Dayal [1986].
- Carey, M., DeWitt, D., and Vandenberg, S. [1988] "A Data Model and Query Language for Exodus," in *SIGMOD* [1988].
- Carey, M., DeWitt, D., Richardson, J., and Shekita, E. [1986a] "Object and File Management in the EXODUS Extensible Database System," in *VLDB* [1986].
- Carey, M., Franklin, M., Livny, M., and Shekita, E. [1991] "Data Caching Tradeoffs in Client-Server DBMS Architectures," in *SIGMOD* [1991].
- Carey, M., and Kossman, D. [1998] "Reducing the breaking distance of an SQL Query Engine," in *VLDB* [1998], pp. 158–169.
- Carlis, J. [1986] "HAS, a Relational Algebra Operator or Divide Is Not Enough to Conquer," in *ICDE* [1986].
- Carlis, J., and March, S. [1984] "A Descriptive Model of Physical Database Design Problems and Solutions," in *ICDE* [1984].
- Carneiro, G., and Vasconcelos, N. [2005] "A Database Centric View of Semantic Image Annotation and Retrieval," in *SIGIR* [2005].
- Carroll, J. M. [1995] **Scenario-Based Design: Envisioning Work and Technology in System Development**, Wiley, 1995.
- Casanova, M., and Vidal, V. [1982] "Toward a Sound View Integration Method," in *PODS* [1982].
- Casanova, M., Fagin, R., and Papadimitriou, C. [1981] "Inclusion Dependencies and Their Interaction with Functional Dependencies," in *PODS* [1981].
- Casanova, M., Furtado, A., and Tuchermann, L. [1991] "A Software Tool for Modular Database Design," *TODS*, 16:2, June 1991.
- Casanova, M., Tuchermann, L., Furtado, A., and Braga, A. [1989] "Optimization of Relational Schemas Containing Inclusion Dependencies," in *VLDB* [1989].
- Castano, S., DeAntonello, V., Fugini, M. G., and Pernici, B. [1998] "Conceptual Schema Analysis: Techniques and Applications," *TODS*, 23:3, September 1998, pp. 286–332.
- Catarci, T., Costabile, M. F., Levialdi, S., and Batini, C. [1997] "Visual Query Systems for Databases: A Survey," **Journal of Visual Languages and Computing**, 8:2, June 1997, pp. 215–260.
- Catarci, T., Costabile, M. F., Santucci, G., and Tarantino, L., eds. [1998] *Proc. Fourth International Workshop on Advanced Visual Interfaces*, ACM Press, 1998.
- Cattell, R. [1991] **Object Data Management: Object-Oriented and Extended Relational Database Systems**, Addison-Wesley, 1991.
- Cattell, R., and Barry, D. K. [2000], **The Object Data Standard: ODMG 3.0**, Morgan Kaufmann, 2000.
- Cattell, R., and Skeen, J. [1992] "Object Operations Benchmark," *TODS*, 17:1, March 1992.
- Cattell, R., ed. [1993] **The Object Database Standard: ODMG-93, Release 1.2**, Morgan Kaufmann, 1993.
- Cattell, R., ed. [1997] **The Object Database Standard: ODMG, Release 2.0**, Morgan Kaufmann, 1997.
- Cattell, R. [2010] "Scalable SQL and NoSQL data stores," **SIGMOD Record**, Vol. 39 Issue 4, 2010.
- Ceri, S., and Fraternali, P. [1997] **Designing Database Applications with Objects and Rules: The IDEA Methodology**, Addison-Wesley, 1997.
- Ceri, S., and Owicki, S. [1983] "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases," *Proc. Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1983.
- Ceri, S., and Pelagatti, G. [1984] "Correctness of Query Execution Strategies in Distributed Databases," *TODS*, 8:4, December 1984.
- Ceri, S., and Pelagatti, G. [1984a] **Distributed Databases: Principles and Systems**, McGraw-Hill, 1984.
- Ceri, S., and Tanca, L. [1987] "Optimization of Systems of Algebraic Equations for Evaluating Datalog Queries," in *VLDB* [1987].
- Ceri, S., Gottlob, G., and Tanca, L. [1990] **Logic Programming and Databases**, Springer-Verlag, 1990.
- Ceri, S., Navathe, S., and Wiederhold, G. [1983] "Distribution Design of Logical Database Schemas," *TSE*, 9:4, July 1983.
- Ceri, S., Negri, M., and Pelagatti, G. [1982] "Horizontal Data Partitioning in Database Design," in *SIGMOD* [1982].
- Cesarini, F., and Soda, G. [1991] "A Dynamic Hash Method with Signature," *TODS*, 16:2, June 1991.
- Chakrabarti, S. [2002] **Mining the Web: Discovering Knowledge from Hypertext Data**. Morgan-Kaufmann, 2002.
- Chakrabarti, S. et al. [1999] "Mining the Web's Link Structure," **Computer** 32:8, August 1999, pp. 60–67.
- Chakravarthy, S. [1990] "Active Database Management Systems: Requirements, State-of-the-Art, and an Evaluation," in *ER Conference* [1990].
- Chakravarthy, S. [1991] "Divide and Conquer: A Basis for Augmenting a Conventional Query Optimizer with Multiple Query Processing Capabilities," in *ICDE* [1991].
- Chakravarthy, S. et al. [1989] "HiPAC: A Research Project in Active, Time Constrained Database Management," Final Technical Report, XAIT-89-02, Xerox Advanced Information Technology, August 1989.
- Chakravarthy, S., Anwar, E., Maugis, L., and Mishra, D. [1994] Design of Sentinel: An Object-oriented DBMS with Event-based Rules, **Information and Software Technology**, 36:9, 1994.



- Chakravarthy, S., Karlapalem, K., Navathe, S., and Tanaka, A. [1993] "Database Supported Co-operative Problem Solving," **International Journal of Intelligent Co-operative Information Systems**, 2:3, September 1993.
- Chakravarthy, U., Grant, J., and Minker, J. [1990] "Logic-Based Approach to Semantic Query Optimization," **TODS**, 15:2, June 1990.
- Chalmers, M., and Chitson, P. [1992] "Bead: Explorations in Information Visualization," *Proc. ACM SIGIR International Conference*, June 1992.
- Chamberlin, D. et al. [1976] "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," **IBM Journal of Research and Development**, 20:6, November 1976.
- Chamberlin, D. et al. [1981] "A History and Evaluation of System R," **CACM**, 24:10, October 1981.
- Chamberlin, D., and Boyce, R. [1974] "SEQUEL: A Structured English Query Language," in *SIGMOD* [1974].
- Chan, C., Ooi, B., and Lu, H. [1992] "Extensible Buffer Management of Indexes," in *VLDB* [1992].
- Chandy, K., Browne, J., Dissley, C., and Uhrig, W. [1975] "Analytical Models for Rollback and Recovery Strategies in Database Systems," **TSE**, 1:1, March 1975.
- Chang, C. [1981] "On the Evaluation of Queries Containing Derived Relations in a Relational Database" in Gallaire et al. [1981].
- Chang, C., and Walker, A. [1984] "PROSQL: A Prolog Programming Interface with SQL/DS," in *EDS* [1984].
- Chang, E., and Katz, R. [1989] "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in Object-Oriented Databases," in *SIGMOD* [1989].
- Chang, F. et al. [2006] "Bigtable: A Distributed Storage System for Structured Data," in *OSDI* [2006].
- Chang, N., and Fu, K. [1981] "Picture Query Languages for Pictorial Databases," **IEEE Computer**, 14:11, November 1981.
- Chang, P., and Myre, W. [1988] "OS/2 EE Database Manager: Overview and Technical Highlights," **IBM Systems Journal**, 27:2, 1988.
- Chang, S., Lin, B., and Walser, R. [1979] "Generalized Zooming Techniques for Pictorial Database Systems," *NCC, AFIPS*, 48, 1979.
- Chatzoglu, P. D., and McCauley, L. A. [1997] "Requirements Capture and Analysis: A Survey of Current Practice," **Requirements Engineering**, 1997, pp. 75–88.
- Chaudhri, A., Rashid, A., and Zicari, R., eds. [2003] **XML Data Management: Native XML and XML-Enabled Database Systems**, Addison-Wesley, 2003.
- Chaudhuri, S., and Dayal, U. [1997] "An Overview of Data Warehousing and OLAP Technology," **SIGMOD Record**, 26:1, March 1997.
- Chaudhuri, S., and Shim, K. [1994] "Including Group-By in Query Optimization," in *VLDB* [1994].
- Chaudhuri, S. et al. [1995] "Optimizing Queries with Materialized Views," in *ICDE* [1995].
- Chen, M., and Yu, P. [1991] "Determining Beneficial Semi-joins for a Join Sequence in Distributed Query Processing," in *ICDE* [1991].
- Chen, M., Han, J., and Yu, P. S., [1996] "Data Mining: An Overview from a Database Perspective," **TKDE**, 8:6, December 1996.
- Chen, P. [1976] "The Entity Relationship Mode—Toward a Unified View of Data," **TODS**, 1:1, March 1976.
- Chen, P., and Patterson, D. [1990]. "Maximizing performance in a striped disk array," in *Proceedings of Symposium on Computer Architecture*, IEEE, New York, 1990.
- Chen, P. et al. [1994] RAID High Performance, Reliable Secondary Storage, **ACM Computing Surveys**, 26:2, 1994.
- Chen, Q., and Kambayashi, Y. [1991] "Nested Relation Based Database Knowledge Representation," in *SIGMOD* [1991].
- Cheng, J. [1991] "Effective Clustering of Complex Objects in Object-Oriented Databases," in *SIGMOD* [1991].
- Cheung, D., et al. [1996] "A Fast and Distributed Algorithm for Mining Association Rules," in *Proc. Int. Conf. on Parallel and Distributed Information Systems*, PDIS [1996].
- Childs, D. [1968] "Feasibility of a Set Theoretical Data Structure—A General Structure Based on a Reconstituted Definition of Relation," *Proc. IFIP Congress*, 1968.
- Chimenti, D. et al. [1987] "An Overview of the LDL System," **IEEE Data Engineering Bulletin**, 10:4, 1987, pp. 52–62.
- Chimenti, D. et al. [1990] "The LDL System Prototype," **TKDE**, 2:1, March 1990.
- Chin, F. [1978] "Security in Statistical Databases for Queries with Small Counts," **TODS**, 3:1, March 1978.
- Chin, F., and Ozsoyoglu, G. [1981] "Statistical Database Design," **TODS**, 6:1, March 1981.
- Chintalapati, R., Kumar, V., and Datta, A. [1997] "An Adaptive Location Management Algorithm for Mobile Computing," *Proc. 22nd Annual Conf. on Local Computer Networks (LCN '97)*, Minneapolis, 1997.
- Chou, H.-T., and DeWitt, D. [1985] "An Evaluation of Buffer Management Strategies for Relational Databases," *VLDB* [1985], pp. 127–141.
- Chou, H.-T., and Kim, W. [1986] "A Unifying Framework for Version Control in a CAD Environment," in *VLDB* [1986], pp. 336–344.
- Christodoulakis, S. et al. [1984] "Development of a Multimedia Information System for an Office Environment," in *VLDB* [1984].

- Christodoulakis, S., and Faloutsos, C. [1986] "Design and Performance Considerations for an Optical Disk-Based Multimedia Object Server," **IEEE Computer**, 19:12, December 1986.
- Chrysanthos, P. [1993] "Transaction Processing in a Mobile Computing Environment," *Proc. IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993, pp. 77–82.
- Chu, W., and Hurley, P. [1982] "Optimal Query Processing for Distributed Database Systems," **IEEE Transactions on Computers**, 31:9, September 1982.
- Ciborra, C., Migliarese, P., and Romano, P. [1984] "A Methodological Inquiry of Organizational Noise in Socio-Technical Systems," **Human Relations**, 37:8, 1984.
- CISCO [2014] Accelerate Application Performance with the Cisco UCS Invicta Series, CISCO White Paper, January 2014.
- Claybrook, B. [1992] **File Management Techniques**, Wiley, 1992.
- Claybrook, B. [1992] **OLTP: OnLine Transaction Processing Systems**, Wiley, 1992.
- Clementini, E., and Di Felice, P. [2000] "Spatial Operators," in **SIGMOD Record** 29:3, 2000, pp. 31–38.
- Clifford, J., and Tansel, A. [1985] "On an Algebra for Historical Relational Databases: Two Views," in **SIGMOD** [1985].
- Clocksin, W. F., and Mellish, C. S. [2003] **Programming in Prolog: Using the ISO Standard**, 5th ed., Springer, 2003.
- Cloudera Inc. [2014] "Impala Performance Update: Now Reaching DBMS-Class Speed," by Justin Erickson et al., (<http://blog.cloudera.com/blog/2014/01/impala-performance-dbms-class-speed/>), January 2014.
- Cockcroft, S. [1997] "A Taxonomy of Spatial Data Integrity Constraints," *GeoInformatica*, 1997, pp. 327–343.
- CODASYL [1978] Data Description Language Journal of Development, Canadian Government Publishing Centre, 1978.
- Codd, E. [1970] "A Relational Model for Large Shared Data Banks," **CACM**, 13:6, June 1970.
- Codd, E. [1971] "A Data Base Sublanguage Founded on the Relational Calculus," *Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control*, November 1971.
- Codd, E. [1972] "Relational Completeness of Data Base Sublanguages," in Rustin [1972].
- Codd, E. [1972a] "Further Normalization of the Data Base Relational Model," in Rustin [1972].
- Codd, E. [1974] "Recent Investigations in Relational Database Systems," *Proc. IFIP Congress*, 1974.
- Codd, E. [1978] "How About Recently? (English Dialog with Relational Data Bases Using Rendezvous Version 1)," in Shneiderman [1978].
- Codd, E. [1979] "Extending the Database Relational Model to Capture More Meaning," **TODS**, 4:4, December 1979.
- Codd, E. [1982] "Relational Database: A Practical Foundation for Productivity," **CACM**, 25:2, December 1982.
- Codd, E. [1985] "Is Your DBMS Really Relational?" and "Does Your DBMS Run By the Rules?," **Computer World**, October 14 and October 21, 1985.
- Codd, E. [1986] "An Evaluation Scheme for Database Management Systems That Are Claimed to Be Relational," in *ICDE* [1986].
- Codd, E. [1990] **Relational Model for Data Management-Version 2**, Addison-Wesley, 1990.
- Codd, E. F., Codd, S. B., and Salley, C. T. [1993] "Providing OLAP (On-Line Analytical Processing) to User Analyst: An IT Mandate," a white paper at [http://www.cs.bgu.ac.il/~dbm031/dw042/Papers/olap\\_to\\_useranalysts\\_wp.pdf](http://www.cs.bgu.ac.il/~dbm031/dw042/Papers/olap_to_useranalysts_wp.pdf), 1993.
- Comer, D. [1979] "The Ubiquitous B-tree," **ACM Computing Surveys**, 11:2, June 1979.
- Comer, D. [2008] **Computer Networks and Internets**, 5th ed., Prentice-Hall, 2008.
- Cooley, R. [2003] "The Use of Web Structure and Content to Identify Subjectively Interesting Web Usage Patterns," **ACM Trans. On Internet Technology**, 3:2, May 2003, pp. 93–116.
- Cooley, R., Mobasher, B., and Srivastava, J. [1997] "Web Mining: Information and Pattern Discovery on the World Wide Web," in *Proc. Ninth IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, November 1997, pp. 558–567.
- Cooley, R., Mobasher, B., and Srivastava, J. [2000] "Automatic personalization based on Web usage mining," **CACM**, 43:8, August 2000.
- Corcho, C., Fernandez-Lopez, M., and Gomez-Perez, A. [2003] "Methodologies, Tools and Languages for Building Ontologies. Where Is Their Meeting Point?," **DKE**, 46:1, July 2003.
- Cormen, T., Leiserson, C. and Rivest, R. [1990] **Introduction to Algorithms**, MIT Press, 1990.
- Cornelio, A., and Navathe, S. [1993] "Applying Active Database Models for Simulation," in *Proceedings of 1993 Winter Simulation Conference*, IEEE, Los Angeles, December 1993.
- Corson, S., and Macker, J. [1999] "Mobile Ad-Hoc Networking: Routing Protocol Performance Issues and Performance Considerations," IETF Request for Comments No. 2501, January 1999, available at [www.ietf.org/rfc/rfc2501.txt](http://www.ietf.org/rfc/rfc2501.txt).
- Cosmadakis, S., Kanellakis, P. C., and Vardi, M. [1990] "Polynomial-time Implication Problems for Unary Inclusion Dependencies," **JACM**, 37:1, 1990, pp. 15–46.

- Covi, L., and Kling, R. [1996] "Organizational Dimensions of Effective Digital Library Use: Closed Rational and Open Natural Systems Models," **Journal of American Society of Information Science (JASIS)**, 47:9, 1996, pp. 672–689.
- Croft, B., Metzler, D., and Strohmman, T. [2009] **Search Engines: Information Retrieval in Practice**, Addison-Wesley, 2009.
- Cruz, I. [1992] "Doodle: A Visual Language for Object-Oriented Databases," in *SIGMOD* [1992].
- Curtice, R. [1981] "Data Dictionaries: An Assessment of Current Practice and Problems," in *VLDB* [1981].
- Cuticchia, A., Fasman, K., Kingsbury, D., Robbins, R., and Pearson, P. [1993] "The GDB Human Genome Database Anno 1993," **Nucleic Acids Research**, 21:13, 1993.
- Czejdo, B., Elmasri, R., Rusinkiewicz, M., and Embley, D. [1987] "An Algebraic Language for Graphical Query Formulation Using an Extended Entity-Relationship Model," *Proc. ACM Computer Science Conference*, 1987.
- Dahl, R., and Bubenko, J. [1982] "IDBD: An Interactive Design Tool for CODASYL DBTG Type Databases," in *VLDB* [1982].
- Dahl, V. [1984] "Logic Programming for Constructive Database Systems," in *EDS* [1984].
- Danforth, S., and Tomlinson, C. [1988] "Type Theories and Object-oriented Programming," **ACM Computing Surveys**, 20:1, 1998, pp. 29–72.
- Das, S. [1992] **Deductive Databases and Logic Programming**, Addison-Wesley, 1992.
- Das, S., Antony, S., Agrawal, D. et al. [2008] "Clouded Data: Comprehending Scalable Data Management Systems," **UCSB CS Technical Report** 2008-18, November 2008.
- Date, C. J. [1983] **An Introduction to Database Systems**, Vol. 2, Addison-Wesley, 1983.
- Date, C. J. [1983a] "The Outer Join," *Proc. Second International Conference on Databases (ICOD-2)*, 1983.
- Date, C. J. [1984] "A Critique of the SQL Database Language," **ACM SIGMOD Record**, 14:3, November 1984.
- Date, C. J. [2001] **The Database Relational Model: A Retrospective Review and Analysis: A Historical Account and Assessment of E. F. Codd's Contribution to the Field of Database Technology**, Addison-Wesley, 2001.
- Date, C. J. [2004] **An Introduction to Database Systems**, 8th ed., Addison-Wesley, 2004.
- Date, C. J., and Darwen, H. [1993] **A Guide to the SQL Standard**, 3rd ed., Addison-Wesley.
- Date C.J. and Fagin, R. [1992] "Simple Conditions for Guaranteeing Higher Normal Forms in Relational Databases," **TODS**, 17:3, 1992.
- Date, C., J. and White, C. [1988] **A Guide to SQL/DS**, Addison-Wesley, 1988.
- Date, C. J., and White, C. [1989] **A Guide to DB2**, 3rd ed., Addison-Wesley, 1989.
- Davies, C. [1973] "Recovery Semantics for a DB/DC System," *Proc. ACM National Conference*, 1973.
- Dayal, U. et al. [1987] "PROBE Final Report," Technical Report CCA-87-02, Computer Corporation of America, December 1987.
- Dayal, U., and Bernstein, P. [1978] "On the Updatability of Relational Views," in *VLDB* [1978].
- Dayal, U., Hsu, M., and Ladin, R. [1991] "A Transaction Model for Long-Running Activities," in *VLDB* [1991].
- DBTG [1971] **Report of the CODASYL Data Base Task Group**, ACM, April 1971.
- DeCandia, G. et al. [2007] "Dynamo: Amazon's Highly Available Key-Value Store," In *SOSP*, 2007.
- Deelman, E., and Chervenak, A. L. [2008] "Data Management Challenges of Data-Intensive Scientific Workflows," in *Proc. IEEE International Symposium on Cluster, Cloud, and Grid Computing*, 2008, pp. 687–692.
- Delcambre, L., Lim, B., and Urban, S. [1991] "Object-Centered Constraints," in *ICDE* [1991].
- DeMarco, T. [1979] **Structured Analysis and System Specification**, Prentice-Hall, 1979.
- DeMers, M. [2002] **Fundamentals of GIS**, John Wiley, 2002.
- DeMichiel, L. [1989] "Performing Operations Over Mismatched Domains," in *ICDE* [1989].
- Denning, D. [1980] "Secure Statistical Databases with Random Sample Queries," **TODS**, 5:3, September 1980.
- Denning, D. E., and Denning, P. J. [1979] "Data Security," **ACM Computing Surveys**, 11:3, September 1979, pp. 227–249.
- Denning, D. et al. [1987] "A Multi-level Relational Data Model," in *Proc. IEEE Symp. On Security and Privacy*, 1987, pp. 196–201.
- Deshpande, A. [1989] "An Implementation for Nested Relational Databases," Technical Report, Ph.D. dissertation, Indiana University, 1989.
- Devor, C., and Weeldreyer, J. [1980] "DDTS: A Testbed for Distributed Database Research," *Proc. ACM Pacific Conference*, 1980.
- DeWitt, D. et al. [1984] "Implementation Techniques for Main Memory Databases," in *SIGMOD* [1984].
- DeWitt, D. et al. [1990] "The Gamma Database Machine Project," **TKDE**, 2:1, March 1990.
- DeWitt, D., Fattersack, P., Maier, D., and Velez, F. [1990] "A Study of Three Alternative Workstation Server Architectures for Object-Oriented Database Systems," in *VLDB* [1990].
- Dhawan, C. [1997] **Mobile Computing**, McGraw-Hill, 1997.
- Di, S. M. [2005] **Distributed Data Management in Grid Environments**, Wiley, 2005.

- Dietrich, B. L. et al. [2014] **Analytics Across the Enterprise: How IBM Realizes Business Value from Big Data and Analytics**, IBM Press (Pearson plc), 2014, 192 pp.
- Dietrich, S., Friesen, O., and Calliss, W. [1999] "On Deductive and Object Oriented Databases: The VALIDITY Experience," Technical Report, Arizona State University, 1999.
- Diffie, W., and Hellman, M. [1979] "Privacy and Authentication," **Proceedings of the IEEE**, 67:3, March 1979, pp. 397–429.
- Dimitrova, N. [1999] "Multimedia Content Analysis and Indexing for Filtering and Retrieval Applications," **Information Science**, Special Issue on Multimedia Informing Technologies, Part 1, 2:4, 1999.
- Dipert, B., and Levy, M. [1993] **Designing with Flash Memory**, Annabooks, 1993.
- Dittrich, K. [1986] "Object-Oriented Database Systems: The Notion and the Issues," in Dittrich and Dayal [1986].
- Dittrich, K., and Dayal, U., eds. [1986] *Proc. International Workshop on Object-Oriented Database Systems*, IEEE CS, Pacific Grove, CA, September 1986.
- Dittrich, K., Kotz, A., and Mulle, J. [1986] "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases," in **ACM SIGMOD Record**, 15:3, 1986.
- DKE [1997] Special Issue on Natural Language Processing, **DKE**, 22:1, 1997.
- Dodd, G. [1969] "APL—A Language for Associative Data Handling in PL/I," *Proc. Fall Joint Computer Conference*, AFIPS, 29, 1969.
- Dodd, G. [1969] "Elements of Data Management Systems," **ACM Computing Surveys**, 1:2, June 1969.
- Dogac, A. [1998] Special Section on Electronic Commerce, **ACM SIGMOD Record**, 27:4, December 1998.
- Dogac, A., Ozsu, M. T., Biliris, A., and Sellis, T., eds. [1994] **Advances in Object-oriented Databases Systems**, NATO ASI Series. Series F: Computer and Systems Sciences, Vol. 130, Springer-Verlag, 1994.
- Dos Santos, C., Neuhold, E., and Furtado, A. [1979] "A Data Type Approach to the Entity-Relationship Model," in *ER Conference* [1979].
- Du, D., and Tong, S. [1991] "Multilevel Extendible Hashing: A File Structure for Very Large Databases," **TKDE**, 3:3, September 1991.
- Du, H., and Ghanta, S. [1987] "A Framework for Efficient IC/VLSI CAD Databases," in *ICDE* [1987].
- Dumas, P. et al. [1982] "MOBILE-Burotique: Prospects for the Future," in Naffah [1982].
- Dumpala, S., and Arora, S. [1983] "Schema Translation Using the Entity-Relationship Approach," in *ER Conference* [1983].
- Dunham, M., and Helal, A. [1995] "Mobile Computing and Databases: Anything New?" **ACM SIGMOD Record**, 24:4, December 1995.
- Dwyer, S. et al. [1982] "A Diagnostic Digital Imaging System," *Proc. IEEE CS Conference on Pattern Recognition and Image Processing*, June 1982.
- Eastman, C. [1987] "Database Facilities for Engineering Design," **Proceedings of the IEEE**, 69:10, October 1981.
- EDS [1984] **Expert Database Systems**, Kerschberg, L., ed. (*Proc. First International Workshop on Expert Database Systems*, Kiawah Island, SC, October 1984), Benjamin/Cummings, 1986.
- EDS [1986] **Expert Database Systems**, Kerschberg, L., ed. (*Proc. First International Conference on Expert Database Systems*, Charleston, SC, April 1986), Benjamin/Cummings, 1987.
- EDS [1988] **Expert Database Systems**, Kerschberg, L., ed. (*Proc. Second International Conference on Expert Database Systems*, Tysons Corner, VA, April 1988), Benjamin/Cummings.
- Eick, C. [1991] "A Methodology for the Design and Transformation of Conceptual Schemas," in *VLDB* [1991].
- ElAbbad, A., and Toueg, S. [1988] "The Group Paradigm for Concurrency Control," in *SIGMOD* [1988].
- ElAbbad, A., and Toueg, S. [1989] "Maintaining Availability in Partitioned Replicated Databases," **TODS**, 14:2, June 1989.
- Ellis, C., and Nutt, G. [1980] "Office Information Systems and Computer Science," **ACM Computing Surveys**, 12:1, March 1980.
- Elmagarmid A. K., ed. [1992] **Database Transaction Models for Advanced Applications**, Morgan Kaufmann, 1992.
- Elmagarmid, A., and Helal, A. [1988] "Supporting Updates in Heterogeneous Distributed Database Systems," in *ICDE* [1988], pp. 564–569.
- Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewicz, M. [1990] "A Multidatabase Transaction Model for Interbase," in *VLDB* [1990].
- Elmasri, R., and Larson, J. [1985] "A Graphical Query Facility for ER Databases," in *ER Conference* [1985].
- Elmasri, R., and Wiederhold, G. [1979] "Data Model Integration Using the Structural Model," in *SIGMOD* [1979].
- Elmasri, R., and Wiederhold, G. [1980] "Structural Properties of Relationships and Their Representation," **NCC**, AFIPS, 49, 1980.
- Elmasri, R., and Wiederhold, G. [1981] "GORDAS: A Formal, High-Level Query Language for the Entity-Relationship Model," in *ER Conference* [1981].
- Elmasri, R., and Wu, G. [1990] "A Temporal Model and Query Language for ER Databases," in *ICDE* [1990].
- Elmasri, R., and Wu, G. [1990a] "The Time Index: An Access Structure for Temporal Data," in *VLDB* [1990].



- Elmasri, R., James, S., and Kouramajian, V. [1993] "Automatic Class and Method Generation for Object-Oriented Databases," *Proc. Third International Conference on Deductive and Object-Oriented Databases (DOOD-93)*, Phoenix, AZ, December 1993.
- Elmasri, R., Kouramajian, V., and Fernando, S. [1993] "Temporal Database Modeling: An Object-Oriented Approach," *CIKM*, November 1993.
- Elmasri, R., Larson, J., and Navathe, S. [1986] "Schema Integration Algorithms for Federated Databases and Logical Database Design," Honeywell CSDD, Technical Report CSC-86-9: 8212, January 1986.
- Elmasri, R., Srinivas, P., and Thomas, G. [1987] "Fragmentation and Query Decomposition in the ECR Model," in *ICDE* [1987].
- Elmasri, R., Weeldreyer, J., and Hevner, A. [1985] "The Category Concept: An Extension to the Entity-Relationship Model," *DKE*, 1:1, May 1985.
- Engelbart, D., and English, W. [1968] "A Research Center for Augmenting Human Intellect," *Proc. Fall Joint Computer Conference*, AFIPS, December 1968.
- Epstein, R., Stonebraker, M., and Wong, E. [1978] "Distributed Query Processing in a Relational Database System," in *SIGMOD* [1978].
- ER Conference [1979] **Entity-Relationship Approach to Systems Analysis and Design**, Chen, P., ed. (*Proc. First International Conference on Entity-Relationship Approach*, Los Angeles, December 1979), North-Holland, 1980.
- ER Conference [1981] **Entity-Relationship Approach to Information Modeling and Analysis**, Chen, P., eds. (*Proc. Second International Conference on Entity-Relationship Approach*, Washington, October 1981), Elsevier Science, 1981.
- ER Conference [1983] **Entity-Relationship Approach to Software Engineering**, Davis, C., Jajodia, S., Ng, P., and Yeh, R., eds. (*Proc. Third International Conference on Entity-Relationship Approach*, Anaheim, CA, October 1983), North-Holland, 1983.
- ER Conference [1985] *Proc. Fourth International Conference on Entity-Relationship Approach*, Liu, J., ed., Chicago, October 1985, IEEE CS.
- ER Conference [1986] *Proc. Fifth International Conference on Entity-Relationship Approach*, Spaccapietra, S., ed., Dijon, France, November 1986, Express-Tirages.
- ER Conference [1987] *Proc. Sixth International Conference on Entity-Relationship Approach*, March, S., ed., New York, November 1987.
- ER Conference [1988] *Proc. Seventh International Conference on Entity-Relationship Approach*, Batini, C., ed., Rome, November 1988.
- ER Conference [1989] *Proc. Eighth International Conference on Entity-Relationship Approach*, Lochovsky, F., ed., Toronto, October 1989.
- ER Conference [1990] *Proc. Ninth International Conference on Entity-Relationship Approach*, Kangassalo, H., ed., Lausanne, Switzerland, September 1990.
- ER Conference [1991] *Proc. Tenth International Conference on Entity-Relationship Approach*, Teorey, T., ed., San Mateo, CA, October 1991.
- ER Conference [1992] *Proc. Eleventh International Conference on Entity-Relationship Approach*, Pernul, G., and Tjoa, A., eds., Karlsruhe, Germany, October 1992.
- ER Conference [1993] *Proc. Twelfth International Conference on Entity-Relationship Approach*, Elmasri, R., and Kouramajian, V., eds., Arlington, TX, December 1993.
- ER Conference [1994] *Proc. Thirteenth International Conference on Entity-Relationship Approach*, Loucopoulos, P., and Theodoulidis, B., eds., Manchester, England, December 1994.
- ER Conference [1995] *Proc. Fourteenth International Conference on ER-OO Modeling*, Papazougrou, M., and Tari, Z., eds., Brisbane, Australia, December 1995.
- ER Conference [1996] *Proc. Fifteenth International Conference on Conceptual Modeling*, Thalheim, B., ed., Cottbus, Germany, October 1996.
- ER Conference [1997] *Proc. Sixteenth International Conference on Conceptual Modeling*, Embley, D., ed., Los Angeles, October 1997.
- ER Conference [1998] *Proc. Seventeenth International Conference on Conceptual Modeling*, Ling, T.-K., ed., Singapore, November 1998.
- ER Conference [1999] *Proc. Eighteenth Conference on Conceptual Modeling*, Akoka, J., Bouzeghoub, M., Comyn-Wattiau, I., Métais, E., (eds.): Paris, France, LNCS 1728, Springer, 1999.
- ER Conference [2000] *Proc. Nineteenth Conference on Conceptual Modeling*, Laender, A., Liddle, S., Storey, V., (eds.), Salt Lake City, LNCS 1920, Springer, 2000.
- ER Conference [2001] *Proc. Twentieth Conference on Conceptual Modeling*, Kunii, H., Jajodia, S., Solveberg, A., (eds.), Yokohama, Japan, LNCS 2224, Springer, 2001.
- ER Conference [2002] *Proc. 21st Int. Conference on Conceptual Modeling*, Spaccapietra, S., March, S., Kambayashi, Y., (eds.), Tampere, Finland, LNCS 2503, Springer, 2002.
- ER Conference [2003] *Proc. 22nd Int. Conference on Conceptual Modeling*, Song, I.-Y., Liddle, S., Ling, T.-W., Scheuermann, P., (eds.), Tampere, Finland, LNCS 2813, Springer, 2003.
- ER Conference [2004] *Proc. 23rd Int. Conference on Conceptual Modeling*, Atzeni, P., Chu, W., Lu, H., Zhou, S.,

- Ling, T.-W., (eds.), Shanghai, China, **LNCS 3288**, Springer, 2004.
- ER Conference [2005] *Proc. 24th Int. Conference on Conceptual Modeling*, Delacambre, L.M.L., Kop, C., Mayr, H., Mylopoulos, J., Pastor, O., (eds.), Klagenfurt, Austria, **LNCS 3716**, Springer, 2005.
- ER Conference [2006] *Proc. 25th Int. Conference on Conceptual Modeling*, Embley, D., Olive, A., Ram, S. (eds.), Tucson, AZ, **LNCS 4215**, Springer, 2006.
- ER Conference [2007] *Proc. 26th Int. Conference on Conceptual Modeling*, Parent, C., Schewe, K.-D., Storey, V., Thalheim, B. (eds.), Auckland, New Zealand, **LNCS 4801**, Springer, 2007.
- ER Conference [2008] *Proc. 27th Int. Conference on Conceptual Modeling*, Li, Q., Spaccapietra, S., Yu, E. S. K., Olive, A. (eds.), Barcelona, Spain, **LNCS 5231**, Springer, 2008.
- ER Conference [2009] *Proc. 28th Int. Conference on Conceptual Modeling*, Laender, A., Castano, S., Dayal, U., Casati, F., de Oliveira (eds.), Gramado, RS, Brazil, **LNCS 5829**, Springer, 2009.
- ER Conference [2010] *Proc. 29th Int. Conference on Conceptual Modeling*, Parsons, J. et al. (eds.), Vancouver, Canada, **LNCS 6412**, Springer, 2010.
- ER Conference [2011] *Proc. 30th Int. Conference on Conceptual Modeling*, Jeusfeld, M. Delcambre, L., and Ling, Tok Wang (eds.), Brussels, Belgium, **LNCS 6998**, Springer, 2011.
- ER Conference [2012] *Proc. 31st Int. Conference on Conceptual Modeling*, Atzeni, P., Cheung, D.W., and Ram, Sudha (eds.), Florence, Italy, **LNCS 7532**, Springer, 2012.
- ER Conference [2013] *Proc. 32nd Int. Conference on Conceptual Modeling*, Ng, Wilfred, Storey, V., and Trujillo, J. (eds.), Hong Kong, China, **LNCS 8217**, Springer, 2013.
- ER Conference [2014] *Proc. 33rd Int. Conference on Conceptual Modeling*, Yu, Eric, Dobbie, G., Jarke, M., Purao, S. (eds.), Atlanta, USA, **LNCS 8824**, Springer, 2014.
- ER Conference [2015] *Proc. 34th Int. Conference on Conceptual Modeling*, Stockholm, Sweden, **LNCS Springer**, forthcoming.
- Erl, T. et al. [2013] **Cloud Computing: Concepts, Technology and Architecture**, Prentice Hall, 2013, 489 pp.
- ESRI [2009] "The Geodatabase: Modeling and Managing Spatial Data" in **ArcNews**, 30:4, ESRI, Winter 2008/2009.
- Ester, M., Kriegel, H.-P., and Jorg, S., [2001] "Algorithms and Applications for Spatial Data Mining," in **Research Monograph in GIS**, CRC Press, [2001].
- Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. [1996]. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *KDD*, 1996, AAAI Press, pp. 226–231.
- Eswaran, K., and Chamberlin, D. [1975] "Functional Specifications of a Subsystem for Database Integrity," in *VLDB* [1975].
- Eswaran, K., Gray, J., Lorie, R., and Traiger, I. [1976] "The Notions of Consistency and Predicate Locks in a Data Base System," **CACM**, 19:11, November 1976.
- Etzioni, O. [1996] "The World-Wide Web: quagmire or gold mine?" **CACM**, 39:11, November 1996, pp. 65–68.
- Everett, G., Dissly, C., and Hardgrave, W. [1971] **RFMS User Manual**, TRM-16, Computing Center, University of Texas at Austin, 1981.
- Fagin, R. [1977] "Multivalued Dependencies and a New Normal Form for Relational Databases," **TODS**, 2:3, September 1977.
- Fagin, R. [1979] "Normal Forms and Relational Database Operators," in *SIGMOD* [1979].
- Fagin, R. [1981] "A Normal Form for Relational Databases That Is Based on Domains and Keys," **TODS**, 6:3, September 1981.
- Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. [1979] "Extendible Hashing—A Fast Access Method for Dynamic Files," **TODS**, 4:3, September 1979.
- Falcone, S., and Paton, N. [1997]. "Deductive Object-Oriented Database Systems: A Survey," *Proc. 3rd International Workshop Rules in Database Systems (RIDS '97)*, Skovde, Sweden, June 1997.
- Faloutsos, C. [1996] **Searching Multimedia Databases by Content**, Kluwer, 1996.
- Faloutsos, C. et al. [1994] "Efficient and Effective Querying by Image Content," **Journal of Intelligent Information Systems**, 3:4, 1994.
- Faloutsos, G., and Jagadish, H. [1992] "On B-Tree Indices for Skewed Distributions," in *VLDB* [1992].
- Fan, J., Gao, Y., Luo, H. and Xu, G. [2004] "Automatic Image Annotation by Using Concept-sensitive Salient Objects for Image Content Representation," in *SIGIR*, 2004.
- Farag, W., and Teorey, T. [1993] "FunBase: A Function-based Information Management System," *CIKM*, November 1993.
- Farahmand, F., Navathe, S., Sharp, G., Enslow, P. [2003] "Managing Vulnerabilities of Information Systems to Security Incidents," *Proc. ACM 5th International Conference on Electronic Commerce, ICEC 2003*, Pittsburgh, PA, September 2003, pp. 348–354.
- Farahmand, F., Navathe, S., Sharp, G., Enslow, P., "A Management Perspective on Risk of Security Threats to Information Systems," **Journal of Information Technology & Management**, Vol. 6, pp. 203–225, 2005.
- Fayyad, U., Piatetsky-Shapiro, G., Smyth, P., Uthurusamy, R. [1997] **Advances in Knowledge Discovery and Data Mining**, MIT Press, 1997.

- Fekete, A., O'Neil, E., and O'Neil, P. [2004] "A Read-only Transaction Anomaly Under Snapshot Isolation," **SIGMOD Record**, 33:3, 2004, pp. 12–14.
- Fekete, A. et al. [2005] "Making Snapshot Isolation Serializable," **ACM TODS**, 30:2, 2005, pp. 492–528.
- Fellbaum, C., ed. [1998] **WordNet: An Electronic Lexical Database**, MIT Press, 1998.
- Fensel, D. [2000] "The Semantic Web and Its Languages," **IEEE Intelligent Systems**, Vol. 15, No. 6, Nov./Dec. 2000, pp. 67–73.
- Fensel, D. [2003]: **Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce**, 2nd ed., Springer-Verlag, Berlin, 2003.
- Fernandez, E., Summers, R., and Wood, C. [1981] **Database Security and Integrity**, Addison-Wesley, 1981.
- Ferrier, A., and Stangret, C. [1982] "Heterogeneity in the Distributed Database Management System SIRIUS-DELTA," in *VLDB* [1982].
- Ferrucci, D. et al. "Building Watson: An overview of the DeepQA project." **AI Magazine** 31:3, 2010, pp. 59–79.
- Fishman, D. et al. [1987] "IRIS: An Object-Oriented DBMS," **TOIS**, 5:1, 1987, pp. 48–69.
- Flickner, M. et al. [1995] "Query by Image and Video Content: The QBIC System," **IEEE Computer**, 28:9, September 1995, pp. 23–32.
- Flynn, J., and Pitts, T. [2000] **Inside ArcINFO 8**, 2nd ed., On Word Press, 2000.
- Folk, M. J., Zoellick, B., and Riccardi, G. [1998] **File Structures: An Object Oriented Approach with C++**, 3rd ed., Addison-Wesley, 1998.
- Fonseca, F., Egenhofer, M., Davis, C. and Câmara, G. [2002] "Semantic Granularity in Ontology-Driven Geographic Information Systems," in **Annals of Mathematics and Artificial Intelligence** 36:1–2, pp. 121–151.
- Ford, D., and Christodoulakis, S. [1991] "Optimizing Random Retrievals from CLV Format Optical Disks," in *VLDB* [1991].
- Ford, D., Blakeley, J., and Bannon, T. [1993] "Open OODB: A Modular Object-Oriented DBMS," in *SIGMOD* [1993].
- Foreman, G., and Zahorjan, J. [1994] "The Challenges of Mobile Computing," **IEEE Computer**, April 1994.
- Fotouhi, F., Grosky, W., Stanchev, P. [2007], eds., *Proc. of the First ACM Workshop on Many Faces of the Multimedia Semantics, MS 2007*, Augsburg Germany, September 2007.
- Fowler, M., and Scott, K. [2000] **UML Distilled**, 2nd ed., Addison-Wesley, 2000.
- Franaszek, P., Robinson, J., and Thomasian, A. [1992] "Concurrency Control for High Contention Environments," **TODS**, 17:2, June 1992.
- Frank, A. [2003] "A linguistically justified proposal for a spatio-temporal ontology," a position paper in *Proc. COSIT03- Int. Conf. on Spatial Information Theory*, Ittingen, Switzerland, LNCS 2825, September 2003.
- Franklin, F. et al. [1992] "Crash Recovery in Client-Server EXODUS," in *SIGMOD* [1992].
- Franks, B. [2012] **Taming the Big Data Tidal Wave**, Wiley, 2012, 294 pp.
- Fraternali, P. [1999] Tools and Approaches for Data Intensive Web Applications: A Survey, *ACM Computing Surveys*, 31:3, September 1999.
- Frenkel, K. [1991] "The Human Genome Project and Informatics," **CACM**, November 1991.
- Friesen, O., Gauthier-Villars, G., Lefelorre, A., and Vieille, L., "Applications of Deductive Object-Oriented Databases Using DEL," in Ramakrishnan (1995).
- Friis-Christensen, A., Tryfona, N., and Jensen, C. S. [2001] "Requirements and Research Issues in Geographic Data Modeling," *Proc. 9th ACM International Symposium on Advances in Geographic Information Systems*, 2001.
- Fugini, M., Castano, S., Martella G., and Samarati, P. [1995] **Database Security**, ACM Press and Addison-Wesley, 1995.
- Furtado, A. [1978] "Formal Aspects of the Relational Model," **Information Systems**, 3:2, 1978.
- Gadia, S. [1988] "A Homogeneous Relational Model and Query Language for Temporal Databases," **TODS**, 13:4, December 1988.
- Gait, J. [1988] "The Optical File Cabinet: A Random-Access File System for Write-Once Optical Disks," **IEEE Computer**, 21:6, June 1988.
- Galindo-Legaria, C. and Joshi, M. [2001] "Orthogonal Optimization of Subqueries and Aggregation," in *SIGMOD* [2001].
- Galindo-Legaria, C., Sefani, S., and Waas, F. [2004] "Query Processing for SQL Updates," in *SIGMOD* [2004], pp. 844–849.
- Gallaire, H., and Minker, J., eds. [1978] **Logic and Databases**, Plenum Press, 1978.
- Gallaire, H., Minker, J., and Nicolas, J. [1984] "Logic and Databases: A Deductive Approach," **ACM Computing Surveys**, 16:2, June 1984.
- Gallaire, H., Minker, J., and Nicolas, J., eds. [1981] **Advances in Database Theory**, Vol. 1, Plenum Press, 1981.
- Gamal-Eldin, M., Thomas, G., and Elmasri, R. [1988] "Integrating Relational Databases with Support for Updates," *Proc. International Symposium on Databases in Parallel and Distributed Systems*, IEEE CS, December 1988.
- Gane, C., and Sarson, T. [1977] **Structured Systems Analysis: Tools and Techniques, Improved Systems Technologies**, 1977.

- Gangopadhyay, A., and Adam, N. [1997] **Database Issues in Geographic Information Systems**, Kluwer Academic Publishers, 1997.
- Garcia-Molina, H. [1982] "Elections in Distributed Computing Systems," **IEEE Transactions on Computers**, 31:1, January 1982.
- Garcia-Molina, H. [1983] "Using Semantic Knowledge for Transaction Processing in a Distributed Database," **TODS**, 8:2, June 1983.
- Garcia-Molina, H., Ullman, J., and Widom, J. [2000] **Database System Implementation**, Prentice-Hall, 2000.
- Garcia-Molina, H., Ullman, J., and Widom, J. [2009] **Database Systems: The Complete Book**, 2nd ed., Prentice-Hall, 2009.
- Gartner [2014a] **Hype Cycle for Information Infrastructure**, by Mark Beyer and Roxanne Edjlali, August 2014, Gartner Press, 110 pp.
- Gartner [2014b] "The Logical Data Warehouse Will be a Key Scenario for Using Data Federation," by Eric Thoo and Ted Friedman, Gartner, September 2012, 6 pp.
- Gedik, B., and Liu, L. [2005] "Location Privacy in Mobile Systems: A Personalized Anonymization Model," in *ICDCS*, 2005, pp. 620–629.
- Gehani, N., Jagdish, H., and Shmueli, O. [1992] "Composite Event Specification in Active Databases: Model and Implementation," in *VLDB* [1992].
- Geman, S., and Geman, D. [1984]. "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. PAMII-6, No. 6, November 1984, pp. 721–741.
- Georgakopoulos, D., Rusinkiewicz, M., and Sheth, A. [1991] "On Serializability of Multidatabase Transactions Through Forced Local Conflicts," in *ICDE* [1991].
- Gerritsen, R. [1975] "A Preliminary System for the Design of DBTG Data Structures," **CACM**, 18:10, October 1975.
- Ghemawat, S., Gobioff, H., and Leung, S. [2003] "The Google File System," in *SOSP* [2003].
- Ghosh, S. [1984] "An Application of Statistical Databases in Manufacturing Testing," in *ICDE* [1984].
- Ghosh, S. [1986] "Statistical Data Reduction for Manufacturing Testing," in *ICDE* [1986].
- Gibson, G. et al. [1997] "File Server Scaling with Network-Attached Secure Disks." Sigmetrics, 1997.
- Gifford, D. [1979] "Weighted Voting for Replicated Data," *SOSP*, 1979.
- Gladney, H. [1989] "Data Replicas in Distributed Information Services," **TODS**, 14:1, March 1989.
- Gogolla, M., and Hohenstein, U. [1991] "Towards a Semantic View of an Extended Entity-Relationship Model," **TODS**, 16:3, September 1991.
- Goldberg, A., and Robson, D. [1989] **Smalltalk-80: The Language**, Addison-Wesley, 1989.
- Goldfine, A., and Konig, P. [1988] *A Technical Overview of the Information Resource Dictionary System (IRDS)*, 2nd ed., NBS IR 88-3700, National Bureau of Standards.
- Goodchild, M. F. [1992] "Geographical Information Science," **International Journal of Geographical Information Systems**, 1992, pp. 31–45.
- Goodchild, M. F. [1992a] "Geographical Data Modeling," **Computers & Geosciences** 18:4, 1992, pp. 401–408.
- Gordillo, S., and Balaguer, F. [1998] "Refining an Object-oriented GIS Design Model: Topologies and Field Data," *Proc. 6th ACM International Symposium on Advances in Geographic Information Systems*, 1998.
- Gotlieb, L. [1975] "Computing Joins of Relations," in *SIGMOD* [1975].
- Graefe, G. [1993] "Query Evaluation Techniques for Large Databases," **ACM Computing Surveys**, 25:2, June 1993.
- Graefe, G., and DeWitt, D. [1987] "The EXODUS Optimizer Generator," in *SIGMOD* [1987].
- Graefe, G., and McKenna, W. [1993] "The Volcano Optimizer Generator," in *ICDE* [1993], pp. 209–218.
- Graefe, G. [1995] "The Cascades Framework for Query Optimization," *Data Engineering Bulletin*, 18:3, 1995, pp. 209–218.
- Gravano, L., and Garcia-Molina, H. [1997] "Merging Ranks from Heterogeneous Sources," in *VLDB* [1997].
- Gray, J. [1978] "Notes on Data Base Operating Systems," in Bayer, Graham, and Seegmuller [1978].
- Gray, J. [1981] "The Transaction Concept: Virtues and Limitations," in *VLDB* [1981].
- Gray, J., and Reuter, A. [1993] **Transaction Processing: Concepts and Techniques**, Morgan Kaufmann, 1993.
- Gray, J., Helland, P., O'Neil, P., and Shasha, D. [1993] "The Dangers of Replication and a Solution," *SIGMOD* [1993].
- Gray, J., Horst, B., and Walker, M. [1990] "Parity Striping of Disk Arrays: Low-Cost Reliable Storage with Acceptable Throughput," in *VLDB* [1990], pp. 148–161.
- Gray, J., Lorie, R., and Putzolu, G. [1975] "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in Nijssen [1975].
- Gray, J., McJones, P., and Blasgen, M. [1981] "The Recovery Manager of the System R Database Manager," **ACM Computing Surveys**, 13:2, June 1981.
- Griffiths, P., and Wade, B. [1976] "An Authorization Mechanism for a Relational Database System," **TODS**, 1:3, September 1976.
- Grochowski, E., and Hoyt, R. F. [1996] "Future Trends in Hard Disk Drives," **IEEE Transactions on Magnetics**, 32:3, May 1996.



- Grosky, W. [1994] "Multimedia Information Systems," in *IEEE Multimedia*, 1:1, Spring 1994.
- Grosky, W. [1997] "Managing Multimedia Information in Database Systems," in *CACM*, 40:12, December 1997.
- Grosky, W., Jain, R., and Mehrotra, R., eds. [1997] **The Handbook of Multimedia Information Management**, Prentice-Hall PTR, 1997.
- Gruber, T. [1995] "Toward principles for the design of ontologies used for knowledge sharing," *International Journal of Human-Computer Studies*, 43:5–6, Nov./Dec. 1995, pp. 907–928.
- Gupta, R. and Horowitz E. [1992] **Object Oriented Databases with Applications to Case, Networks and VLSI CAD**, Prentice-Hall, 1992.
- Güting, R. [1994] "An Introduction to Spatial Database Systems," in *VLDB* [1994].
- Guttman, A. [1984] "R-Trees: A Dynamic Index Structure for Spatial Searching," in *SIGMOD* [1984].
- Gwayer, M. [1996] **Oracle Designer/2000 Web Server Generator Technical Overview** (version 1.3.2), Technical Report, Oracle Corporation, September 1996.
- Gyssens, M., Paredaens, J., and Van Gucht, D. [1990] "A graph-oriented object model for database end-user interfaces," in *SIGMOD* [1990].
- Haas, P., and Swami, A. [1995] "Sampling-based Selectivity Estimation for Joins Using Augmented Frequent Value Statistics," in *ICDE* [1995].
- Haas, P., Naughton, J., Seshadri, S., and Stokes, L. [1995] "Sampling-based Estimation of the Number of Distinct Values of an Attribute," in *VLDB* [1995].
- Hachem, N., and Berra, P. [1992] "New Order Preserving Access Methods for Very Large Files Derived from Linear Hashing," *TKDE*, 4:1, February 1992.
- Hadoop [2014] Hadoop Wiki at <http://hadoop.apache.org/>
- Hadzilacos, V. [1983] "An Operational Model for Database System Reliability," in *Proceedings of SIGACT-SIGMOD Conference*, March 1983.
- Hadzilacos, V. [1988] "A Theory of Reliability in Database Systems," *JACM*, 35:1, 1986.
- Haerder, T., and Reuter, A. [1983] "Principles of Transaction Oriented Database Recovery—A Taxonomy," *ACM Computing Surveys*, 15:4, September 1983, pp. 287–318.
- Haerder, T., and Rothermel, K. [1987] "Concepts for Transaction Recovery in Nested Transactions," in *SIGMOD* [1987].
- Hakonarson, H., Gulcher, J., and Stefansson, K. [2003]. "deCODE genetics, Inc." **Pharmacogenomics Journal**, 2003, pp. 209–215.
- Halfond, W., and Orso, A. [2005] "AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks," in *Proc. IEEE and ACM Int. Conf. on Automated Software Engineering (ASE 2005)*, November 2005, pp. 174–183.
- Halfond, W., Viegas, J., and Orso, A. [2006] "A Classification of SQL Injection Attacks and Countermeasures," in *Proc. Int. Symposium on Secure Software Engineering*, March 2006.
- Hall, P. [1976] "Optimization of a Single Relational Expression in a Relational Data Base System," *IBM Journal of Research and Development*, 20:3, May 1976.
- Hamilton, G., Catteli, R., and Fisher, M. [1997] **JDBC Database Access with Java—A Tutorial and Annotated Reference**, Addison-Wesley, 1997.
- Hammer, M., and McLeod, D. [1975] "Semantic Integrity in a Relational Data Base System," in *VLDB* [1975].
- Hammer, M., and McLeod, D. [1981] "Database Description with SDM: A Semantic Data Model," *TODS*, 6:3, September 1981.
- Hammer, M., and Sarin, S. [1978] "Efficient Monitoring of Database Assertions," in *SIGMOD* [1978].
- Han, J., Kamber, M., and Pei, J. [2005] **Data Mining: Concepts and Techniques**, 2nd ed., Morgan Kaufmann, 2005.
- Han, Y., Jiang, C. and Luo, X. [2004] "A Study of Concurrency Control in Web-Based Distributed Real-Time Database System Using Extended Time Petri Nets," *Proc. Int. Symposium on Parallel Architectures, Algorithms, and Networks*, 2004, pp. 67–72.
- Han, J., Pei, J., and Yin, Y. [2000] "Mining Frequent Patterns without Candidate Generation," in *SIGMOD* [2000].
- Hanson, E. [1992] "Rule Condition Testing and Action Execution in Ariel," in *SIGMOD* [1992].
- Hardgrave, W. [1980] "Ambiguity in Processing Boolean Queries on TDMS Tree Structures: A Study of Four Different Philosophies," *TSE*, 6:4, July 1980.
- Hardgrave, W. [1984] "BOLT: A Retrieval Language for Tree-Structured Database Systems," in *Tou* [1984].
- Harel, D., [1987] "Statecharts: A Visual Formulation for Complex Systems," in *Science of Computer Programming*, 8:3, June 1987, pp. 231–274.
- Harman, D. [1992] "Evaluation Issues in Information Retrieval," *Information Processing and Management*, 28:4, pp. 439–440.
- Harrington, J. [1987] **Relational Database Management for Microcomputer: Design and Implementation**, Holt, Rinehart, and Winston, 1987.
- Harris, L. [1978] "The ROBOT System: Natural Language Processing Applied to Data Base Query," *Proc. ACM National Conference*, December 1978.
- Harth, A., Hose, K., and Schenkel, R. [2014] **Linked Data Management**, Chapman and Hall, CRC Press, 2014, 576 pp.
- Haskin, R., and Lorie, R. [1982] "On Extending the Functions of a Relational Database System," in *SIGMOD* [1982].

- Hasse, C., and Weikum, G. [1991] "A Performance Evaluation of Multi-Level Transaction Management," in *VLDB* [1991].
- Hayes-Roth, F., Waterman, D., and Lenat, D., eds. [1983] **Building Expert Systems**, Addison-Wesley, 1983.
- Hayne, S., and Ram, S. [1990] "Multi-User View Integration System: An Expert System for View Integration," in *ICDE* [1990].
- Hecht, R., and Jablonski, S. [2011] "NOSQL Evaluation, A Use Case Oriented Survey," in *Int. Conf. on Cloud and Service Computing*, IEEE, 2011, pp. 336–341.
- Heiler, S., and Zdonick, S. [1990] "Object Views: Extending the Vision," in *ICDE* [1990].
- Heiler, S., Hardhvalal, S., Zdonik, S., Blaustein, B., and Rosenthal, A. [1992] "A Flexible Framework for Transaction Management in Engineering Environment," in Elmagarmid [1992].
- Helal, A., Hu, T., Elmasri, R., and Mukherjee, S. [1993] "Adaptive Transaction Scheduling," *CIKM*, November 1993.
- Held, G., and Stonebraker, M. [1978] "B-Trees Reexamined," *CACM*, 21:2, February 1978.
- Henriksen, C., Lauzon, J. P., and Morehouse, S. [1994] "Open Geodata Access Through Standards," *Standard-View Archive*, 1994, 2:3, pp. 169–174.
- Henschen, L., and Naqvi, S. [1984] "On Compiling Queries in Recursive First-Order Databases," *JACM*, 31:1, January 1984.
- Hernandez, H., and Chan, E. [1991] "Constraint-Time-Maintainable BCNF Database Schemes," *TODS*, 16:4, December 1991.
- Herot, C. [1980] "Spatial Management of Data," *TODS*, 5:4, December 1980.
- Hevner, A., and Yao, S. [1979] "Query Processing in Distributed Database Systems," *TSE*, 5:3, May 1979.
- Hinneburg, A., and Gabriel, H.-H., [2007] "DENCLUE 2.0: Fast Clustering Based on Kernel Density Estimation," in *Proc. IDA'2007: Advances in Intelligent Data Analysis VII, 7th International Symposium on Intelligent Data Analysis*, Ljubljana, Slovenia, September 2007, LNCS 4723, Springer, 2007.
- Hoffer, J. [1982] "An Empirical Investigation with Individual Differences in Database Models," *Proc. Third International Information Systems Conference*, December 1982.
- Hoffer, J., Prescott, M., and Topi, H. [2009] **Modern Database Management**, 9th ed., Prentice-Hall, 2009.
- Holland, J. [1975] **Adaptation in Natural and Artificial Systems**, University of Michigan Press, 1975.
- Holsapple, C., and Whinston, A., eds. [1987] **Decision Support Systems Theory and Application**, Springer-Verlag, 1987.
- Holt, R. C. [1972] "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, 4:3, pp. 179–196.
- Holtzman J. M., and Goodman D. J., eds. [1993] **Wireless Communications: Future Directions**, Kluwer, 1993.
- Horowitz, B. [1992] "A Run-Time Execution Model for Referential Integrity Maintenance," in *ICDE* [1992], pp. 548–556.
- Hortonworks, Inc. [2014a] "Benchmarking Apache Hive 13 for Enterprise Hadoop," by Carter Shanklin, a Hortonworks Blog (<http://hortonworks.com/blog/benchmarking-apache-hive-13-enterprise-hadoop/>), June 2014.
- Hortonworks, Inc. [2014b] "Best Practices—Selecting Apache Hadoop Hardware," at [http://docs.hortonworks.com/HDP2Alpha/index.htm#Hardware\\_Recommendations\\_for\\_Hadoop.htm](http://docs.hortonworks.com/HDP2Alpha/index.htm#Hardware_Recommendations_for_Hadoop.htm)
- Howson, C. and P. Urbach, P. [1993] **Scientific Reasoning: The Bayesian Approach**, Open Court Publishing, December 1993.
- Hsiao, D., and Kamel, M. [1989] "Heterogeneous Databases: Proliferation, Issues, and Solutions," *TKDE*, 1:1, March 1989.
- Hsu, A., and Imielinsky, T. [1985] "Integrity Checking for Multiple Updates," in *SIGMOD* [1985].
- Hsu, M., and Zhang, B. [1992] "Performance Evaluation of Cautious Waiting," *TODS*, 17:3, pp. 477–512.
- Hull, R., and King, R. [1987] "Semantic Database Modeling: Survey, Applications, and Research Issues," *ACM Computing Surveys*, 19:3, September 1987.
- Huxhold, W. [1991] **An Introduction to Urban Geographic Information Systems**, Oxford University Press, 1991.
- IBM [1978] **QBE Terminal Users Guide**, Form Number SH20-2078-0.
- IBM [1992] **Systems Application Architecture Common Programming Interface Database Level 2 Reference**, Document Number SC26-4798-01.
- ICDE* [1984] *Proc. IEEE CS International Conference on Data Engineering*, Shuey, R., ed., Los Angeles, CA, April 1984.
- ICDE* [1986] *Proc. IEEE CS International Conference on Data Engineering*, Wiederhold, G., ed., Los Angeles, February 1986.
- ICDE* [1987] *Proc. IEEE CS International Conference on Data Engineering*, Wah, B., ed., Los Angeles, February 1987.
- ICDE* [1988] *Proc. IEEE CS International Conference on Data Engineering*, Carlis, J., ed., Los Angeles, February 1988.
- ICDE* [1989] *Proc. IEEE CS International Conference on Data Engineering*, Shuey, R., ed., Los Angeles, February 1989.
- ICDE* [1990] *Proc. IEEE CS International Conference on Data Engineering*, Liu, M., ed., Los Angeles, February 1990.
- ICDE* [1991] *Proc. IEEE CS International Conference on Data Engineering*, Cercone, N., and Tsuchiya, M., eds., Kobe, Japan, April 1991.

- ICDE [1992] *Proc. IEEE CS International Conference on Data Engineering*, Golshani, F., ed., Phoenix, AZ, February 1992.
- ICDE [1993] *Proc. IEEE CS International Conference on Data Engineering*, Elmagarmid, A., and Neuhold, E., eds., Vienna, Austria, April 1993.
- ICDE [1994] *Proc. IEEE CS International Conference on Data Engineering*, Houston, TX, February 1994.
- ICDE [1995] *Proc. IEEE CS International Conference on Data Engineering*, Yu, P. S., and Chen, A. L. A., eds., Taipei, Taiwan, 1995.
- ICDE [1996] *Proc. IEEE CS International Conference on Data Engineering*, Su, S. Y. W., ed., New Orleans, 1996.
- ICDE [1997] *Proc. IEEE CS International Conference on Data Engineering*, Gray, W. A., and Larson, P. A., eds., Birmingham, England, 1997.
- ICDE [1998] *Proc. IEEE CS International Conference on Data Engineering*, Orlando, FL, February 1998.
- ICDE [1999] *Proc. IEEE CS International Conference on Data Engineering*, Sydney, Australia, March 1999.
- ICDE [2000] *Proc. IEEE CS International Conference on Data Engineering*, San Diego, CA, February-March 2000.
- ICDE [2001] *Proc. IEEE CS International Conference on Data Engineering*, Heidelberg, Germany, April 2001.
- ICDE [2002] *Proc. IEEE CS International Conference on Data Engineering*, San Jose, CA, February-March 2002.
- ICDE [2003] *Proc. IEEE CS International Conference on Data Engineering*, Dayal, U., Ramamritham, K., and Vijayaraman, T. M., eds., Bangalore, India, March 2003.
- ICDE [2004] *Proc. IEEE CS International Conference on Data Engineering*, Boston, MA, March-April 2004.
- ICDE [2005] *Proc. IEEE CS International Conference on Data Engineering*, Tokyo, Japan, April 2005.
- ICDE [2006] *Proc. IEEE CS International Conference on Data Engineering*, Liu, L., Reuter, A., Whang, K.-Y., and Zhang, J., eds., Atlanta, GA, April 2006.
- ICDE [2007] *Proc. IEEE CS International Conference on Data Engineering*, Istanbul, Turkey, April 2007.
- ICDE [2008] *Proc. IEEE CS International Conference on Data Engineering*, Cancun, Mexico, April 2008.
- ICDE [2009] *Proc. IEEE CS International Conference on Data Engineering*, Shanghai, China, March-April 2009.
- ICDE [2010] *Proc. IEEE CS International Conference on Data Engineering*, Long Beach, CA, March 2010.
- ICDE [2011] *Proc. IEEE CS International Conference on Data Engineering*, Hannover, Germany, April 2011.
- ICDE [2012] *Proc. IEEE CS International Conference on Data Engineering*, Kementsietsidis, A., and Antonio Vaz Salles, M., eds., Washington, D.C., April 2012.
- ICDE [2013] *Proc. IEEE CS International Conference on Data Engineering*, Jensen, C., Jermaine, C., and Zhou, Xiaofang, eds., Brisbane, Australia, April 2013.
- ICDE [2014] *Proc. IEEE CS International Conference on Data Engineering*, Cruz, Isabel F. et al., eds., Chicago, March-April 2014.
- ICDE [2015] *Proc. IEEE CS International Conference on Data Engineering*, Seoul Korea, April 2015, forthcoming.
- IGES [1983] International Graphics Exchange Specification Version 2, National Bureau of Standards, U.S. Department of Commerce, January 1983.
- Imielinski, T., and Badrinath, B. [1994] "Mobile Wireless Computing: Challenges in Data Management," *CACM*, 37:10, October 1994.
- Imielinski, T., and Lipski, W. [1981] "On Representing Incomplete Information in a Relational Database," in *VLDB* [1981].
- Indulska, M., and Orłowska, M. E. [2002] "On Aggregation Issues in Spatial Data Management," (ACM International Conference Proceeding Series) *Proc. Thirteenth Australasian Conference on Database Technologies*, Melbourne, 2002, pp. 75–84.
- Informix [1998] "Web Integration Option for Informix Dynamic Server," available at [www.informix.com](http://www.informix.com).
- Inmon, W. H. [1992] **Building the Data Warehouse**, Wiley, 1992.
- Inmon, W., Strauss, D., and Neushloss, G. [2008] **DW 2.0: The Architecture for the Next Generation of Data Warehousing**, Morgan Kaufmann, 2008.
- Integrity [2004] "An Introduction to SQL Injection Attacks for Oracle Developers," Integrity, April 2004, available at [www.net-security.org/dl/articles/Integrity Intro-to-SQLInjectionAttacks.pdf](http://www.net-security.org/dl/articles/Integrity%20Intro-to-SQLInjectionAttacks.pdf).
- Internet Engineering Task Force (IETF) [1999] "An Architecture Framework for High Speed Mobile Ad Hoc Network," in *Proc. 45th IETF Meeting*, Oslo, Norway, July 1999, available at [www.ietf.org/proceedings/99/jul/](http://www.ietf.org/proceedings/99/jul/).
- Ioannidis, Y., and Kang, Y. [1990] "Randomized Algorithms for Optimizing Large Join Queries," in *SIGMOD* [1990].
- Ioannidis, Y., and Kang, Y. [1991] "Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization," in *SIGMOD* [1991].
- Ioannidis, Y., and Wong, E. [1988] "Transforming Non-Linear Recursion to Linear Recursion," in *EDS* [1988].
- Iossophidis, J. [1979] "A Translator to Convert the DDL of ERM to the DDL of System 2000," in *ER Conference* [1979].
- Irani, K., Purkayastha, S., and Teorey, T. [1979] "A Designer for DBMS-Processable Logical Database Structures," in *VLDB* [1979].
- Iyer et al. [2004] "A Framework for Efficient Storage Security in RDBMSs," in *EDBT*, 2004, pp. 147–164.
- Jacobson, I., Booch, G., and Rumbaugh, J. [1999] **The Unified Software Development Process**, Addison-Wesley, 1999.

- Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G. [1992] **Object-Oriented Software Engineering: A Use Case Driven Approach**, Addison-Wesley, 1992.
- Jagadish, H. [1989] "Incorporating Hierarchy in a Relational Model of Data," in *SIGMOD* [1989].
- Jagadish, H. [1997] "Content-based Indexing and Retrieval," in Grosky et al. [1997].
- Jajodia, S., Ammann, P., McCollum, C. D., "Surviving Information Warfare Attacks," *IEEE Computer*, 32:4, April 1999, pp. 57–63.
- Jajodia, S., and Kogan, B. [1990] "Integrating an Object-oriented Data Model with Multilevel Security," *Proc. IEEE Symposium on Security and Privacy*, May 1990, pp. 76–85.
- Jajodia, S., and Mutchler, D. [1990] "Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database," *TODS*, 15:2, June 1990.
- Jajodia, S., and Sandhu, R. [1991] "Toward a Multilevel Secure Relational Data Model," in *SIGMOD* [1991].
- Jajodia, S., Ng, P., and Springsteel, F. [1983] "The Problem of Equivalence for Entity-Relationship Diagrams," *TSE*, 9:5, September 1983.
- Jardine, D., ed. [1977] **The ANSI/SPARC DBMS Model**, North-Holland, 1977.
- Jarke, M., and Koch, J. [1984] "Query Optimization in Database Systems," *ACM Computing Surveys*, 16:2, June 1984.
- Jensen, C. et al. [1994] "A Glossary of Temporal Database Concepts," *ACM SIGMOD Record*, 23:1, March 1994.
- Jensen, C., and Snodgrass, R. [1992] "Temporal Specialization," in *ICDE* [1992].
- Jensen, C. et al. [2001] "Location-based Services: A Database Perspective," *Proc. ScanGIS Conference*, 2001, pp. 59–68.
- Jhingran, A., and Khedkar, P. [1992] "Analysis of Recovery in a Database System Using a Write-ahead Log Protocol," in *SIGMOD* [1992].
- Jing, J., Helal, A., and Elmagarmid, A. [1999] "Client-server Computing in Mobile Environments," *ACM Computing Surveys*, 31:2, June 1999.
- Johnson, T., and Shasha, D. [1993] "The Performance of Current B-Tree Algorithms," *TODS*, 18:1, March 1993.
- Jorwekar, S. et al. [2007] "Automating the Detection of Snapshot Isolation Anomalies," in *VLDB* [2007], pp. 1263–1274.
- Joshi, J., Aref, W., Ghafoor, A., and Spafford, E. [2001] "Security Models for Web-Based Applications," *CACM*, 44:2, February 2001, pp. 38–44.
- Jukic, N., Vrbsky, S., and Nestorov, S. [2013] **Database Systems: Introduction to Databases and Data Warehouses**, Prentice Hall, 2013, 408 pp.
- Jung, I.Y., and Yeom, H.Y. [2008] "An efficient and transparent transaction management based on the data workflow of HVEM DataGrid," *Proc. Challenges of Large Applications in Distributed Environments*, 2008, pp. 35–44.
- Kaefer, W., and Schoening, H. [1992] "Realizing a Temporal Complex-Object Data Model," in *SIGMOD* [1992].
- Kamel, I., and Faloutsos, C. [1993] "On Packing R-trees," *CIKM*, November 1993.
- Kamel, N., and King, R. [1985] "A Model of Data Distribution Based on Texture Analysis," in *SIGMOD* [1985].
- Kappel, G., and Schrefl, M. [1991] "Object/Behavior Diagrams," in *ICDE* [1991].
- Karlapalem, K., Navathe, S. B., and Ammar, M. [1996] "Optimal Redesign Policies to Support Dynamic Processing of Applications on a Distributed Relational Database System," *Information Systems*, 21:4, 1996, pp. 353–367.
- Karolchik, D. et al. [2003] "The UCSC Genome Browser Database," in *Nucleic Acids Research*, 31:1, January 2003.
- Katz, R. [1985] **Information Management for Engineering Design: Surveys in Computer Science**, Springer-Verlag, 1985.
- Katz, R., and Wong, E. [1982] "Decompiling CODASYL DML into Relational Queries," *TODS*, 7:1, March 1982.
- Kavis, M. [2014] **Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)**, Wiley, 224 pp.
- KDD [1996] *Proc. Second International Conference on Knowledge Discovery in Databases and Data Mining*, Portland, Oregon, August 1996.
- Ke, Y., and Sukthankar, R. [2004] "PCA-SIFT: A More Distinctive Representation for Local Image Descriptors," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2004.
- Kedem, Z., and Silberschatz, A. [1980] "Non-Two Phase Locking Protocols with Shared and Exclusive Locks," in *VLDB* [1980].
- Keller, A. [1982] "Updates to Relational Database Through Views Involving Joins," in Scheuermann [1982].
- Kemp, K. [1993]. "Spatial Databases: Sources and Issues," in **Environmental Modeling with GIS**, Oxford University Press, New York, 1993.
- Kemper, A., and Wallrath, M. [1987] "An Analysis of Geometric Modeling in Database Systems," *ACM Computing Surveys*, 19:1, March 1987.
- Kemper, A., Lockemann, P., and Wallrath, M. [1987] "An Object-Oriented Database System for Engineering Applications," in *SIGMOD* [1987].
- Kemper, A., Moerkotte, G., and Steinbrunn, M. [1992] "Optimizing Boolean Expressions in Object Bases," in *VLDB* [1992].
- Kent, W. [1978] **Data and Reality**, North-Holland, 1978.
- Kent, W. [1979] "Limitations of Record-Based Information Models," *TODS*, 4:1, March 1979.



- Kent, W. [1991] "Object-Oriented Database Programming Languages," in *VLDB* [1991].
- Kerschberg, L., Ting, P., and Yao, S. [1982] "Query Optimization in Star Computer Networks," *TODS*, 7:4, December 1982.
- Ketabchi, M. A., Mathur, S., Risch, T., and Chen, J. [1990] "Comparative Analysis of RDBMS and OODBMS: A Case Study," *IEEE International Conference on Manufacturing*, 1990.
- Khan, L. [2000] "Ontology-based Information Selection," Ph.D. dissertation, University of Southern California, August 2000.
- Khoshafian, S., and Baker A. [1996] **Multimedia and Imaging Databases**, Morgan Kaufmann, 1996.
- Khoshafian, S., Chan, A., Wong, A., and Wong, H.K.T. [1992] **Developing Client Server Applications**, Morgan Kaufmann, 1992.
- Khoury, M. [2002] "Epidemiology and the Continuum from Genetic Research to Genetic Testing," in **American Journal of Epidemiology**, 2002, pp. 297–299.
- Kifer, M., and Lozinskii, E. [1986] "A Framework for an Efficient Implementation of Deductive Databases," *Proc. Sixth Advanced Database Symposium*, Tokyo, August 1986.
- Kim W. [1995] **Modern Database Systems: The Object Model, Interoperability, and Beyond**, ACM Press, Addison-Wesley, 1995.
- Kim, P. [1996] "A Taxonomy on the Architecture of Database Gateways for the Web," Working Paper TR-96-U-10, Chungnam National University, Taejon, Korea (available from <http://grigg.chungnam.ac.kr/projects/UniWeb>).
- Kim, S.-H., Yoon, K.-J., and Kweon, I.-S. [2006] "Object Recognition Using a Generalized Robust Invariant Feature and Gestalt's Law of Proximity and Similarity," in *Proc. Conf. on Computer Vision and Pattern Recognition Workshop (CVPRW '06)*, 2006.
- Kim, W. [1982] "On Optimizing an SQL-like Nested Query," *TODS*, 3:3, September 1982.
- Kim, W. [1989] "A Model of Queries for Object-Oriented Databases," in *VLDB* [1989].
- Kim, W. [1990] "Object-Oriented Databases: Definition and Research Directions," *TKDE*, 2:3, September 1990.
- Kim, W. et al. [1987] "Features of the ORION Object-Oriented Database System," Microelectronics and Computer Technology Corporation, Technical Report ACA-ST-308-87, September 1987.
- Kim, W., and Lochovsky, F., eds. [1989] **Object-oriented Concepts, Databases, and Applications**, ACM Press, Frontier Series, 1989.
- Kim, W., Garza, J., Ballou, N., and Woelk, D. [1990] "Architecture of the ORION Next-Generation Database System," *TKDE*, 2:1, 1990, pp. 109–124.
- Kim, W., Reiner, D. S., and Batory, D., eds. [1985] **Query Processing in Database Systems**, Springer-Verlag, 1985.
- Kimball, R. [1996] **The Data Warehouse Toolkit**, Wiley, Inc. 1996.
- King, J. [1981] "QUIST: A System for Semantic Query Optimization in Relational Databases," in *VLDB* [1981].
- Kitsuregawa, M., Nakayama, M., and Takagi, M. [1989] "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method," in *VLDB* [1989].
- Kleinberg, J. M. [1999] "Authoritative sources in a hyper-linked environment," *JACM* 46:5, September 1999, pp. 604–632.
- Klimbie, J., and Koffeman, K., eds. [1974] **Data Base Management**, North-Holland, 1974.
- Klug, A. [1982] "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *JACM*, 29:3, July 1982.
- Knuth, D. [1998] **The Art of Computer Programming, Vol. 3: Sorting and Searching**, 2nd ed., Addison-Wesley, 1998.
- Kogelnik, A. [1998] "Biological Information Management with Application to Human Genome Data," Ph.D. dissertation, Georgia Institute of Technology and Emory University, 1998.
- Kogelnik, A. et al. [1998] "MITOMAP: A human mitochondrial genome database—1998 update," **Nucleic Acids Research**, 26:1, January 1998.
- Kogelnik, A., Navathe, S., Wallace, D. [1997] "GENOME: A system for managing Human Genome Project Data." *Proceedings of Genome Informatics '97, Eighth Workshop on Genome Informatics*, Tokyo, Japan, Sponsor: Human Genome Center, University of Tokyo, December 1997.
- Kohler, W. [1981] "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," **ACM Computing Surveys**, 13:2, June 1981.
- Konsynski, B., Bracker, L., and Bracker, W. [1982] "A Model for Specification of Office Communications," **IEEE Transactions on Communications**, 30:1, January 1982.
- Kooi, R. P., [1980] **The Optimization of Queries in Relational Databases**, Ph.D. Dissertation, Case Western Reserve University, 1980: pp. 1–159.
- Koperski, K., and Han, J. [1995] "Discovery of Spatial Association Rules in Geographic Information Databases," in *Proc. SSD'1995, 4th Int. Symposium on Advances in Spatial Databases*, Portland, Maine, LNCS 951, Springer, 1995.
- Korfhage, R. [1991] "To See, or Not to See: Is that the Query?" in *Proc. ACM SIGIR International Conference*, June 1991.

- Korth, H. [1983] "Locking Primitives in a Database System," **JACM**, 30:1, January 1983.
- Korth, H., Levy, E., and Silberschatz, A. [1990] "A Formal Approach to Recovery by Compensating Transactions," in *VLDB* [1990].
- Kosala, R., and Blockeel, H. [2000] "Web Mining Research: a Survey," **SIGKDD Explorations**, 2:1, June 2000, pp. 1–15.
- Kotz, A., Dittrich, K., Mülle, J. [1988] "Supporting Semantic Rules by a Generalized Event/Trigger Mechanism," in *VLDB* [1988].
- Kotz, S., Balakrishnan, N., and Johnson, N. L. [2000] "Dirichlet and Inverted Dirichlet Distributions," in **Continuous Multivariate Distributions: Models and Applications, Vol. 1**, 2<sup>nd</sup> Ed., John Wiley, 2000.
- Krishnamurthy, R., and Naqvi, S. [1989] "Non-Deterministic Choice in Datalog," *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, Jerusalem, June 1989.
- Krishnamurthy, R., Litwin, W., and Kent, W. [1991] "Language Features for Interoperability of Databases with Semantic Discrepancies," in *SIGMOD* [1991].
- Krovetz, R., and Croft B. [1992] "Lexical Ambiguity and Information Retrieval" in **TOIS**, 10, April 1992.
- Kubiatowicz, J. et al., [2000] "OceanStore: An Architecture for Global-Scale Persistent Storage," *ASPLOS* 2000.
- Kuhn, R. M., Karolchik, D., Zweig, et al. [2009] "The UCSC Genome Browser Database: update 2009," **Nucleic Acids Research**, 37:1, January 2009.
- Kulkarni K. et al., "Introducing Reference Types and Cleaning Up SQL3's Object Model," *ISO WG3 Report X3H2-95-456*, November 1995.
- Kumar, A. [1991] "Performance Measurement of Some Main Memory Recovery Algorithms," in *ICDE* [1991].
- Kumar, A., and Segev, A. [1993] "Cost and Availability Tradeoffs in Replicated Concurrency Control," **TODS**, 18:1, March 1993.
- Kumar, A., and Stonebraker, M. [1987] "Semantics Based Transaction Management Techniques for Replicated Data," in *SIGMOD* [1987].
- Kumar, D. [2007a]. "Genomic medicine: a new frontier of medicine in the twenty first century," **Genomic Medicine**, 2007, pp. 3–7.
- Kumar, D. [2007b]. "Genome mirror—2006," **Genomic Medicine**, 2007, pp. 87–90.
- Kumar, V., and Han, M., eds. [1992] **Recovery Mechanisms in Database Systems**, Prentice-Hall, 1992.
- Kumar, V., and Hsu, M. [1998] **Recovery Mechanisms in Database Systems**, Prentice-Hall (PTR), 1998.
- Kumar, V., and Song, H. S. [1998] **Database Recovery**, Kluwer Academic, 1998.
- Kung, H., and Robinson, J. [1981] "Optimistic Concurrency Control," **TODS**, 6:2, June 1981.
- Lacroix, M., and Pirotte, A. [1977a] "Domain-Oriented Relational Languages," in *VLDB* [1977].
- Lacroix, M., and Pirotte, A. [1977b] "ILL: An English Structured Query Language for Relational Data Bases," in Nijssen [1977].
- Lai, M.-Y., and Wilkinson, W. K. [1984] "Distributed Transaction Management in Jasmin," in *VLDB* [1984].
- Lamb, C. et al. [1991] "The ObjectStore Database System," in **CACM**, 34:10, October 1991, pp. 50–63.
- Lamport, L. [1978] "Time, Clocks, and the Ordering of Events in a Distributed System," **CACM**, 21:7, July 1978.
- Lander, E. [2001] "Initial Sequencing and Analysis of the Genome," **Nature**, 409:6822, 2001.
- Langerak, R. [1990] "View Updates in Relational Databases with an Independent Scheme," **TODS**, 15:1, March 1990.
- Lanka, S., and Mays, E. [1991] "Fully Persistent B1-Trees," in *SIGMOD* [1991].
- Larson, J. [1983] "Bridging the Gap Between Network and Relational Database Management Systems," **IEEE Computer**, 16:9, September 1983.
- Larson, J., Navathe, S., and Elmasri, R. [1989] "Attribute Equivalence and its Use in Schema Integration," **TSE**, 15:2, April 1989.
- Larson, P. [1978] "Dynamic Hashing," **BIT**, 18, 1978.
- Larson, P. [1981] "Analysis of Index-Sequential Files with Overflow Chaining," **TODS**, 6:4, December 1981.
- Lassila, O. [1998] "Web Metadata: A Matter of Semantics," **IEEE Internet Computing**, 2:4, July/August 1998, pp. 30–37.
- Laurini, R., and Thompson, D. [1992] **Fundamentals of Spatial Information Systems**, Academic Press, 1992.
- Lausen G., and Vossen, G. [1997] **Models and Languages of Object Oriented Databases**, Addison-Wesley, 1997.
- Lazebnik, S., Schmid, C., and Ponce, J. [2004] "Semi-Local Affine Parts for Object Recognition," in *Proc. British Machine Vision Conference*, Kingston University, The Institution of Engineering and Technology, U.K., 2004.
- Lee, J., Elmasri, R., and Won, J. [1998] "An Integrated Temporal Data Model Incorporating Time Series Concepts," **DKE**, 24, 1998, pp. 257–276.
- Lehman, P., and Yao, S. [1981] "Efficient Locking for Concurrent Operations on B-Trees," **TODS**, 6:4, December 1981.
- Lehman, T., and Lindsay, B. [1989] "The Starburst Long Field Manager," in *VLDB* [1989].
- Leiss, E. [1982] "Randomizing: A Practical Method for Protecting Statistical Databases Against Compromise," in *VLDB* [1982].
- Leiss, E. [1982a] **Principles of Data Security**, Plenum Press, 1982.

- Lenat, D. [1995] "CYC: A Large-Scale Investment in Knowledge Infrastructure," *CACM* 38:11, November 1995, pp. 32–38.
- Lenzerini, M., and Santucci, C. [1983] "Cardinality Constraints in the Entity Relationship Model," in *ER Conference* [1983].
- Leung, C., Hibler, B., and Mwara, N. [1992] "Picture Retrieval by Content Description," in *Journal of Information Science*, 1992, pp. 111–119.
- Levesque, H. [1984] "The Logic of Incomplete Knowledge Bases," in Brodie et al., Ch. 7 [1984].
- Li, W.-S., Seluk Candan, K., Hirata, K., and Hara, Y. [1998] Hierarchical Image Modeling for Object-based Media Retrieval in *DKE*, 27:2, September 1998, pp. 139–176.
- Lien, E., and Weinberger, P. [1978] "Consistency, Concurrency, and Crash Recovery," in *SIGMOD* [1978].
- Lieuwen, L., and DeWitt, D. [1992] "A Transformation-Based Approach to Optimizing Loops in Database Programming Languages," in *SIGMOD* [1992].
- Lilien, L., and Bhargava, B. [1985] "Database Integrity Block Construct: Concepts and Design Issues," *TSE*, 11:9, September 1985.
- Lin, J., and Dunham, M. H. [1998] "Mining Association Rules," in *ICDE* [1998].
- Lindsay, B. et al. [1984] "Computation and Communication in R\*: A Distributed Database Manager," *TOCS*, 2:1, January 1984.
- Lippman R. [1987] "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, April 1987.
- Lipski, W. [1979] "On Semantic Issues Connected with Incomplete Information," *TODS*, 4:3, September 1979.
- Lipton, R., Naughton, J., and Schneider, D. [1990] "Practical Selectivity Estimation through Adaptive Sampling," in *SIGMOD* [1990].
- Liskov, B., and Zilles, S. [1975] "Specification Techniques for Data Abstractions," *TSE*, 1:1, March 1975.
- Litwin, W. [1980] "Linear Hashing: A New Tool for File and Table Addressing," in *VLDB* [1980].
- Liu, B. [2006] **Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (Data-Centric Systems and Applications)**, Springer, 2006.
- Liu, B. and Chen-Chuan-Chang, K. [2004] "Editorial: Special Issue on Web Content Mining," *SIGKDD Explorations Newsletter* 6:2, December 2004, pp. 1–4.
- Liu, K., and Sunderraman, R. [1988] "On Representing Indefinite and Maybe Information in Relational Databases," in *ICDE* [1988].
- Liu, L., and Meersman, R. [1992] "Activity Model: A Declarative Approach for Capturing Communication Behavior in Object-Oriented Databases," in *VLDB* [1992].
- Lockemann, P., and Knutsen, W. [1968] "Recovery of Disk Contents After System Failure," *CACM*, 11:8, August 1968.
- Longley, P. et al [2001] **Geographic Information Systems and Science**, John Wiley, 2001.
- Lorie, R. [1977] "Physical Integrity in a Large Segmented Database," *TODS*, 2:1, March 1977.
- Lorie, R., and Plouffe, W. [1983] "Complex Objects and Their Use in Design Transactions," in *SIGMOD* [1983].
- Lowe, D. [2004] "Distinctive Image Features from Scale-Invariant Keypoints," *Int. Journal of Computer Vision*, Vol. 60, 2004, pp. 91–110.
- Lozinskii, E. [1986] "A Problem-Oriented Inferential Database System," *TODS*, 11:3, September 1986.
- Lu, H., Mikkilineni, K., and Richardson, J. [1987] "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation," in *ICDE* [1987].
- Lubars, M., Potts, C., and Richter, C. [1993] "A Review of the State of Practice in Requirements Modeling," Proc. IEEE International Symposium on Requirements Engineering, San Diego, CA, 1993.
- Lucyk, B. [1993] **Advanced Topics in DB2**, Addison-Wesley, 1993.
- Luhn, H. P. [1957] "A Statistical Approach to Mechanized Encoding and Searching of Literary Information," *IBM Journal of Research and Development*, 1:4, October 1957, pp. 309–317.
- Lunt, T., and Fernandez, E. [1990] "Database Security," in *SIGMOD Record*, 19:4, pp. 90–97.
- Lunt, T. et al. [1990] "The Seaview Security Model," *IEEE TSE*, 16:6, pp. 593–607.
- Luo, J., and Nascimento, M. [2003] "Content-based Sub-image Retrieval via Hierarchical Tree Matching," in *Proc. ACM Int Workshop on Multimedia Databases*, New Orleans, pp. 63–69.
- Madria, S. et al. [1999] "Research Issues in Web Data Mining," in *Proc. First Int. Conf. on Data Warehousing and Knowledge Discovery* (Mohania, M., and Tjoa, A., eds.) LNCS 1676. Springer, pp. 303–312.
- Madria, S., Baseer, Mohammed, B., Kumar, V., and Bhowmick, S. [2007] "A transaction model and multiversion concurrency control for mobile database systems," *Distributed and Parallel Databases (DPD)*, 22:2–3, 2007, pp. 165–196.
- Maguire, D., Goodchild, M., and Rhind, D., eds. [1997] **Geographical Information Systems: Principles and Applications. Vols. 1 and 2**, Longman Scientific and Technical, New York.
- Mahajan, S., Donahoo, M. J., Navathe, S. B., Ammar, M., Malik, S. [1998] "Grouping Techniques for Update Propagation in Intermittently Connected Databases," in *ICDE* [1998].
- Maier, D. [1983] **The Theory of Relational Databases**, Computer Science Press, 1983.

- Maier, D., and Warren, D. S. [1988] **Computing with Logic**, Benjamin Cummings, 1988.
- Maier, D., Stein, J., Otis, A., and Purdy, A. [1986] "Development of an Object-Oriented DBMS," *OOPSLA*, 1986.
- Malewicz, G. [2010] "Pregel: a system for large-scale graph processing," in *SIGMOD* [2010].
- Malley, C., and Zdonick, S. [1986] "A Knowledge-Based Approach to Query Optimization," in *EDS* [1986].
- Mannila, H., Toivonen, H., and Verkamo, A. [1994] "Efficient Algorithms for Discovering Association Rules," in *KDD-94, AAAI Workshop on Knowledge Discovery in Databases*, Seattle, 1994.
- Manning, C., and Schütze, H. [1999] **Foundations of Statistical Natural Language Processing**, MIT Press, 1999.
- Manning, C., Raghavan, P., and Schütze, H. [2008] **Introduction to Information Retrieval**, Cambridge University Press, 2008.
- Manola, F. [1998] "Towards a Richer Web Object Model," in *ACM SIGMOD Record*, 27:1, March 1998.
- Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A., and Theodoridis, Y. [2005] **R-Trees: Theory and Applications**, Springer, 2005.
- March, S., and Severance, D. [1977] "The Determination of Efficient Record Segmentations and Blocking Factors for Shared Files," *TODS*, 2:3, September 1977.
- Mark, L., Roussopoulos, N., Newsome, T., and Laohapipattana, P. [1992] "Incrementally Maintained Network to Relational Mappings," *Software Practice & Experience*, 22:12, December 1992.
- Markowitz, V., and Raz, Y. [1983] "ERROL: An Entity-Relationship, Role Oriented, Query Language," in *ER Conference* [1983].
- Martin, J., and Odell, J. [2008] **Principles of Object-oriented Analysis and Design**, Prentice-Hall, 2008.
- Martin, J., Chapman, K., and Leben, J. [1989] **DB2-Concepts, Design, and Programming**, Prentice-Hall, 1989.
- Maryanski, F. [1980] "Backend Database Machines," *ACM Computing Surveys*, 12:1, March 1980.
- Masunaga, Y. [1987] "Multimedia Databases: A Formal Framework," *Proc. IEEE Office Automation Symposium*, April 1987.
- Mattison, R., **Data Warehousing: Strategies, Technologies, and Techniques**, McGraw-Hill, 1996.
- Maune, D. F. [2001] **Digital Elevation Model Technologies and Applications: The DEM Users Manual**, ASPRS, 2001.
- McCarty, C. et al. [2005]. "Marshfield Clinic Personalized Medicine Research Project (PMRP): design, methods and recruitment for a large population-based biobank," *Personalized Medicine*, 2005, pp. 49–70.
- McClure, R., and Krüger, I. [2005] "SQL DOM: Compile Time Checking of Dynamic SQL Statements," *Proc. 27th Int. Conf. on Software Engineering*, May 2005.
- McKinsey [2013] **Big data: The next frontier for innovation, competition, and productivity**, McKinsey Global Institute, 2013, 216 pp.
- McLeish, M. [1989] "Further Results on the Security of Partitioned Dynamic Statistical Databases," *TODS*, 14:1, March 1989.
- McLeod, D., and Heimbigner, D. [1985] "A Federated Architecture for Information Systems," *TOOIS*, 3:3, July 1985.
- Mehrotra, S. et al. [1992] "The Concurrency Control Problem in Multidatabases: Characteristics and Solutions," in *SIGMOD* [1992].
- Melton, J. [2003] **Advanced SQL: 1999—Understanding Object-Relational and Other Advanced Features**, Morgan Kaufmann, 2003.
- Melton, J., and Mattos, N. [1996] "An Overview of SQL3—The Emerging New Generation of the SQL Standard, Tutorial No. T5," *VLDB*, Bombay, September 1996.
- Melton, J., and Simon, A. R. [1993] **Understanding the New SQL: A Complete Guide**, Morgan Kaufmann, 1993.
- Melton, J., and Simon, A. R. [2002] **SQL: 1999—Understanding Relational Language Components**, Morgan Kaufmann, 2002.
- Melton, J., Bauer, J., and Kulkarni, K. [1991] "Object ADTs (with improvements for value ADTs)," *ISO WG3 Report X3H2-91-083*, April 1991.
- Menasce, D., Popek, G., and Muntz, R. [1980] "A Locking Protocol for Resource Coordination in Distributed Databases," *TODS*, 5:2, June 1980.
- Mendelzon, A., and Maier, D. [1979] "Generalized Mutual Dependencies and the Decomposition of Database Relations," in *VLDB* [1979].
- Mendelzon, A., Mihaila, G., and Milo, T. [1997] "Querying the World Wide Web," *Journal of Digital Libraries*, 1:1, April 1997.
- Mesnier, M. et al. [2003]. "Object-Based Storage," *IEEE Communications Magazine*, August 2003, pp. 84–90.
- Metais, E., Kedad, Z., Comyn-Wattiau, C., and Bouzeghoub, M., "Using Linguistic Knowledge in View Integration: Toward a Third Generation of Tools," *DKE*, 23:1, June 1998.
- Mihailescu, M., Soundararajan, G., and Amza, C. "MixA-part: Decoupled Analytics for Shared Storage Systems" In *USENIX Conf on File And Storage Technologies (FAST)*, 2013
- Mikkilineni, K., and Su, S. [1988] "An Evaluation of Relational Join Algorithms in a Pipelined Query Processing Environment," *TSE*, 14:6, June 1988.



- Mikolajczyk, K., and Schmid, C. [2005] "A performance evaluation of local descriptors," **IEEE Transactions on PAMI**, 10:27, 2005, pp. 1615–1630.
- Miller, G. A. [1990] "Nouns in WordNet: a lexical inheritance system." in **International Journal of Lexicography** 3:4, 1990, pp. 245–264.
- Miller, H. J., (2004) "Tobler's First Law and Spatial Analysis," *Annals of the Association of American Geographers*, 94:2, 2004, pp. 284–289.
- Milojicic, D. et al. [2002] *Peer-to-Peer Computing*, HP Laboratories Technical Report No. HPL-2002-57, HP Labs, Palo Alto, available at [www.hpl.hp.com/techreports/2002/HPL-2002-57R1.html](http://www.hpl.hp.com/techreports/2002/HPL-2002-57R1.html).
- Minoura, T., and Wiederhold, G. [1981] "Resilient Extended True-Copy Token Scheme for a Distributed Database," **TSE**, 8:3, May 1981.
- Missikoff, M., and Wiederhold, G. [1984] "Toward a Unified Approach for Expert and Database Systems," in **EDS** [1984].
- Mitchell, T. [1997] **Machine Learning**, McGraw-Hill, 1997.
- Mitschang, B. [1989] "Extending the Relational Algebra to Capture Complex Objects," in **VLDB** [1989].
- Moczar, L. [2015] **Enterprise Lucene and Solr**, Addison Wesley, forthcoming, 2015, 496 pp.
- Mohan, C. [1993] "IBM's Relational Database Products: Features and Technologies," in **SIGMOD** [1993].
- Mohan, C. et al. [1992] "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," **TODS**, 17:1, March 1992.
- Mohan, C., and Levine, F. [1992] "ARIES/IM: An Efficient and High-Concurrency Index Management Method Using Write-Ahead Logging," in **SIGMOD** [1992].
- Mohan, C., and Narang, I. [1992] "Algorithms for Creating Indexes for Very Large Tables without Quiescing Updates," in **SIGMOD** [1992].
- Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. [1992] "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," **TODS**, 17:1, March 1992.
- Morris, K. et al. [1987] "YAWN! (Yet Another Window on NAIL!)," in **ICDE** [1987].
- Morris, K., Ullman, J., and VanGelden, A. [1986] "Design Overview of the NAIL! System," *Proc. Third International Conference on Logic Programming*, Springer-Verlag, 1986.
- Morris, R. [1968] "Scatter Storage Techniques," **CACM**, 11:1, January 1968.
- Morsi, M., Navathe, S., and Kim, H. [1992] "An Extensible Object-Oriented Database Testbed," in **ICDE** [1992].
- Moss, J. [1982] "Nested Transactions and Reliable Distributed Computing," *Proc. Symposium on Reliability in Distributed Software and Database Systems*, IEEE CS, July 1982.
- Motro, A. [1987] "Superviews: Virtual Integration of Multiple Databases," **TSE**, 13:7, July 1987.
- Mouratidis, K. et al. [2006] "Continuous nearest neighbor monitoring in road networks," in **VLDB** [2006], pp. 43–54.
- Mukkamala, R. [1989] "Measuring the Effect of Data Distribution and Replication Models on Performance Evaluation of Distributed Systems," in **ICDE** [1989].
- Mumick, I., Finkelstein, S., Pirahesh, H., and Ramakrishnan, R. [1990a] "Magic Is Relevant" in **SIGMOD** [1990].
- Mumick, I., Pirahesh, H., and Ramakrishnan, R. [1990b] "The Magic of Duplicates and Aggregates," in **VLDB** [1990].
- Muralikrishna, M. [1992] "Improved Unnesting Algorithms for Join and Aggregate SQL Queries," in **VLDB** [1992].
- Muralikrishna, M., and DeWitt, D. [1988] "Equi-depth Histograms for Estimating Selectivity Factors for Multi-dimensional Queries," in **SIGMOD** [1988].
- Murthy, A.C. and Vavilapalli, V.K. [2014] **Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2**, Addison Wesley, 2014, 304 pp.
- Mylopoulos, J., Bernstein, P., and Wong, H. [1980] "A Language Facility for Designing Database-Intensive Applications," **TODS**, 5:2, June 1980.
- Naedele, M., [2003] Standards for XML and Web Services Security, **IEEE Computer**, 36:4, April 2003, pp. 96–98.
- Naish, L., and Thom, J. [1983] "The MU-PROLOG Deductive Database," Technical Report 83/10, Department of Computer Science, University of Melbourne, 1983.
- Natan R. [2005] **Implementing Database Security and Auditing: Includes Examples from Oracle, SQL Server, DB2 UDB, and Sybase**, Digital Press, 2005.
- Navathe, S. [1980] "An Intuitive Approach to Normalize Network-Structured Data," in **VLDB** [1980].
- Navathe, S., and Balaraman, A. [1991] "A Transaction Architecture for a General Purpose Semantic Data Model," in **ER** [1991], pp. 511–541.
- Navathe, S. B., Karlapalem, K., and Ra, M. Y. [1996] "A Mixed Fragmentation Methodology for the Initial Distributed Database Design," **Journal of Computers and Software Engineering**, 3:4, 1996.
- Navathe, S. B. et al. [1994] "Object Modeling Using Classification in CANDIDE and Its Application," in Dogac et al. [1994].
- Navathe, S., and Ahmed, R. [1989] "A Temporal Relational Model and Query Language," **Information Sciences**, 47:2, March 1989, pp. 147–175.
- Navathe, S., and Gadgil, S. [1982] "A Methodology for View Integration in Logical Database Design," in **VLDB** [1982].

- Navathe, S., and Kerschberg, L. [1986] "Role of Data Dictionaries in Database Design," **Information and Management**, 10:1, January 1986.
- Navathe, S., and Savasere, A. [1996] "A Practical Schema Integration Facility Using an Object Oriented Approach," in **Multidatabase Systems** (A. Elmagarmid and O. Bukhres, eds.), Prentice-Hall, 1996.
- Navathe, S., and Schkolnick, M. [1978] "View Representation in Logical Database Design," in **SIGMOD** [1978].
- Navathe, S., Ceri, S., Wiederhold, G., and Dou, J. [1984] "Vertical Partitioning Algorithms for Database Design," **TODS**, 9:4, December 1984.
- Navathe, S., Elmasri, R., and Larson, J. [1986] "Integrating User Views in Database Design," **IEEE Computer**, 19:1, January 1986.
- Navathe, S., Patil, U., and Guan, W. [2007] "Genomic and Proteomic Databases: Foundations, Current Status and Future Applications," in **Journal of Computer Science and Engineering**, Korean Institute of Information Scientists and Engineers (KIISE), 1:1, 2007, pp. 1–30.
- Navathe, S., Sashidhar, T., and Elmasri, R. [1984a] "Relationship Merging in Schema Integration," in **VLDB** [1984].
- Negri, M., Pelagatti, S., and Sbatella, L. [1991] "Formal Semantics of SQL Queries," **TODS**, 16:3, September 1991.
- Ng, P. [1981] "Further Analysis of the Entity-Relationship Approach to Database Design," **TSE**, 7:1, January 1981.
- Ngu, A. [1989] "Transaction Modeling," in **ICDE** [1989], pp. 234–241.
- Nicolas, J. [1978] "Mutual Dependencies and Some Results on Undecomposable Relations," in **VLDB** [1978].
- Nicolas, J. [1997] "Deductive Object-oriented Databases, Technology, Products, and Applications: Where Are We?" *Proc. Symposium on Digital Media Information Base (DMIB '97)*, Nara, Japan, November 1997.
- Nicolas, J., Phipps, G., Derr, M., and Ross, K. [1991] "Glue-NAIL!: A Deductive Database System," in **SIGMOD** [1991].
- Niemiec, R. [2008] **Oracle Database 10g Performance Tuning Tips & Techniques**, McGraw Hill Osborne Media, 2008, 967 pp.
- Nievergelt, J. [1974] "Binary Search Trees and File Organization," **ACM Computing Surveys**, 6:3, September 1974.
- Nievergelt, J., Hinterberger, H., and Seveik, K. [1984]. "The Grid File: An Adaptable Symmetric Multikey File Structure," **TODS**, 9:1, March 1984, pp. 38–71.
- Nijssen, G., ed. [1976] **Modelling in Data Base Management Systems**, North-Holland, 1976.
- Nijssen, G., ed. [1977] **Architecture and Models in Data Base Management Systems**, North-Holland, 1977.
- Nwosu, K., Berra, P., and Thuraisingham, B., eds. [1996] **Design and Implementation of Multimedia Database Management Systems**, Kluwer Academic, 1996.
- O'Neil, P., and O'Neil, P. [2001] **Database: Principles, Programming, Performance**, Morgan Kaufmann, 1994.
- Obermarck, R. [1982] "Distributed Deadlock Detection Algorithms," **TODS**, 7:2, June 1982.
- Oh, Y.-C. [1999] "Secure Database Modeling and Design," Ph.D. dissertation, College of Computing, Georgia Institute of Technology, March 1999.
- Ohsuga, S. [1982] "Knowledge Based Systems as a New Interactive Computer System of the Next Generation," in **Computer Science and Technologies**, North-Holland, 1982.
- Olken, F., Jagadish, J. [2003] Management for Integrative Biology," **OMICS: A Journal of Integrative Biology**, 7:1, January 2003.
- Olle, T. [1978] **The CODASYL Approach to Data Base Management**, Wiley, 1978.
- Olle, T., Sol, H., and Verrijn-Stuart, A., eds. [1982] **Information System Design Methodology**, North-Holland, 1982.
- Olston, C. et al. [2008] Pig Latin: A Not-So-Foreign language for Data Processing, in **SIGMOD** [2008].
- Omiecinski, E., and Scheuermann, P. [1990] "A Parallel Algorithm for Record Clustering," **TODS**, 15:4, December 1990.
- Omura, J. K. [1990] "Novel applications of cryptography in digital communications," **IEEE Communications Magazine**, 28:5, May 1990, pp. 21–29.
- O'Neil, P. and Graefe, G., 'Multi-Table Joins Through Bitmapped Join Indices', **SIGMOD Record**, Vol. 24, No. 3, 1995.
- Open GIS Consortium, Inc. [1999] "*OpenGIS® Simple Features Specification for SQL*," Revision 1.1, OpenGIS Project Document 99-049, May 1999.
- Open GIS Consortium, Inc. [2003] "*OpenGIS® Geography Markup Language (GML) Implementation Specification*," Version 3, OGC 02-023r4., 2003.
- Oracle [2005] **Oracle 10, Introduction to LDAP and Oracle Internet Directory** 10g Release 2, Oracle Corporation, 2005.
- Oracle [2007] **Oracle Label Security Administrator's Guide, 11g (release 11.1)**, Part no. B28529-01, Oracle, available at [http://download.oracle.com/docs/cd/B28359\\_01/network.111/b28529/intro.htm](http://download.oracle.com/docs/cd/B28359_01/network.111/b28529/intro.htm).
- Oracle [2008] **Oracle 11 Distributed Database Concepts** 11g Release 1, Oracle Corporation, 2008.
- Oracle [2009] "An Oracle White Paper: Leading Practices for Driving Down the Costs of Managing Your Oracle Identity and Access Management Suite," Oracle, April 2009.
- Osborn, S. L. [1977] "Normal Forms for Relational Databases," Ph.D. dissertation, University of Waterloo, 1977.
- Osborn, S. L. [1989] "The Role of Polymorphism in Schema Evolution in an Object-Oriented Database," **TKDE**, 1:3, September 1989.

- Osborn, S. L. [1979] "Towards a Universal Relation Interface," in *VLDB* [1979].
- Ozsoyoglu, G., Ozsoyoglu, Z., and Matos, V. [1985] "Extending Relational Algebra and Relational Calculus with Set Valued Attributes and Aggregate Functions," *TODS*, 12:4, December 1987.
- Ozsoyoglu, Z., and Yuan, L. [1987] "A New Normal Form for Nested Relations," *TODS*, 12:1, March 1987.
- Ozsu, M. T., and Valduriez, P. [1999] **Principles of Distributed Database Systems**, 2nd ed., Prentice-Hall, 1999.
- Palanisamy, B. et al. [2011] "Purlieus: locality-aware resource allocation for MapReduce in a cloud," In *Proc. ACM/IEEE Int. Conf for High Perf Computing, Networking, Storage and Analysis, (SC)* 2011.
- Palanisamy, B. et al. [2014] "VNCache: Map Reduce Analysis for Cloud-archived Data", *Proc. 14th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, 2014.
- Palanisamy, B., Singh, A., and Liu, Ling, "Cost-effective Resource Provisioning for MapReduce in a Cloud", *IEEE TPDS*, 26:5, May 2015.
- Papadias, D. et al. [2003] "Query Processing in Spatial Network Databases," in *VLDB* [2003] pp. 802–813.
- Papadimitriou, C. [1979] "The Serializability of Concurrent Database Updates," *JACM*, 26:4, October 1979.
- Papadimitriou, C. [1986] **The Theory of Database Concurrency Control**, Computer Science Press, 1986.
- Papadimitriou, C., and Kanellakis, P. [1979] "On Concurrency Control by Multiple Versions," *TODS*, 9:1, March 1974.
- Papazoglou, M., and Valder, W. [1989] **Relational Database Management: A Systems Programming Approach**, Prentice-Hall, 1989.
- Paredaens, J., and Van Gucht, D. [1992] "Converting Nested Algebra Expressions into Flat Algebra Expressions," *TODS*, 17:1, March 1992.
- Parent, C., and Spaccapietra, S. [1985] "An Algebra for a General Entity-Relationship Model," *TSE*, 11:7, July 1985.
- Paris, J. [1986] "Voting with Witnesses: A Consistency Scheme for Replicated Files," in *ICDE* [1986].
- Park, J., Chen, M., and Yu, P. [1995] "An Effective Hash-Based Algorithm for Mining Association Rules," in *SIGMOD* [1995].
- Parker Z., Poe, S., and Vrbsky, S.V. [2013] "Comparing NoSQL MongoDB to an SQL DB," *Proc. 51st ACM Southeast Conference [ACMSE '13]*, Savannah, GA, 2013.
- Paton, A. W., ed. [1999] **Active Rules in Database Systems**, Springer-Verlag, 1999.
- Paton, N. W., and Diaz, O. [1999] Survey of Active Database Systems, *ACM Computing Surveys*, 31:1, 1999, pp. 63–103.
- Patterson, D., Gibson, G., and Katz, R. [1988] "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *SIGMOD* [1988].
- Paul, H. et al. [1987] "Architecture and Implementation of the Darmstadt Database Kernel System," in *SIGMOD* [1987].
- Pavlo, A. et al. [2009] A Comparison of Approaches to Large Scale Data Analysis, in *SIGMOD* [2009].
- Pazandak, P., and Srivastava, J., "Evaluating Object DBMSs for Multimedia," *IEEE Multimedia*, 4:3, pp. 34–49.
- Pazos- Rangel, R. et. al. [2006] "Least Likely to Use: A New Page Replacement Strategy for Improving Database Management System Response Time," in *Proc. CSR 2006: Computer Science- Theory and Applications*, St. Petersburg, Russia, LNCS, Volume 3967, Springer, 2006, pp. 314–323.
- PDES [1991] "A High-Lead Architecture for Implementing a PDES/STEP Data Sharing Environment," Publication Number PT 1017.03.00, PDES Inc., May 1991.
- Pearson, P. et al. [1994] "The Status of Online Mendelian Inheritance in Man (OMIM) Medio 1994" *Nucleic Acids Research*, 22:17, 1994.
- Peckham, J., and Maryanski, F. [1988] "Semantic Data Models," *ACM Computing Surveys*, 20:3, September 1988, pp. 153–189.
- Peng, T. and Tsou, M. [2003] **Internet GIS: Distributed Geographic Information Services for the Internet and Wireless Network**, Wiley, 2003.
- Pfleeger, C. P., and Pfleeger, S. [2007] **Security in Computing**, 4th ed., Prentice-Hall, 2007.
- Phipps, G., Derr, M., and Ross, K. [1991] "Glue-NAIL: A Deductive Database System," in *SIGMOD* [1991].
- Piatetsky-Shapiro, G., and Frawley, W., eds. [1991] **Knowledge Discovery in Databases**, AAAI Press/MIT Press, 1991.
- Pistor P., and Anderson, F. [1986] "Designing a Generalized NF2 Model with an SQL-type Language Interface," in *VLDB* [1986], pp. 278–285.
- Pitoura, E., and Bhargava, B. [1995] "Maintaining Consistency of Data in Mobile Distributed Environments." In *15th ICDCS*, May 1995, pp. 404–413.
- Pitoura, E., and Samaras, G. [1998] **Data Management for Mobile Computing**, Kluwer, 1998.
- Pitoura, E., Bukhres, O., and Elmagarmid, A. [1995] "Object Orientation in Multidatabase Systems," *ACM Computing Surveys*, 27:2, June 1995.
- Polavarapu, N. et al. [2005] "Investigation into Biomedical Literature Screening Using Support Vector Machines," in *Proc. 4th Int. IEEE Computational Systems Bioinformatics Conference (CSB'05)*, August 2005, pp. 366–374.
- Poncelson D. et al. [1999] "CueVideo: Automated Multimedia Indexing and Retrieval," *Proc. 7th ACM Multimedia Conf.*, Orlando, FL, October 1999, p.199.

- Ponniah, P. [2010] **Data Warehousing Fundamentals for IT Professionals**, 2nd Ed., Wiley Interscience, 2010, 600pp.
- Poosala, V., Ioannidis, Y., Haas, P., and Shekita, E. [1996] "Improved Histograms for Selectivity Estimation of Range Predicates," in *SIGMOD* [1996].
- Porter, M. F. [1980] "An algorithm for suffix stripping," *Program*, 14:3, pp. 130–137.
- Ports, D.R.K. and Grittner, K. [2012] "Serializable Snapshot Isolation in PostgreSQL," *Proceedings of VLDB*, 5:12, 2012, pp. 1850–1861.
- Potter, B., Sinclair, J., and Till, D. [1996] **An Introduction to Formal Specification and Z**, 2nd ed., Prentice-Hall, 1996.
- Prabhakaran, B. [1996] **Multimedia Database Management Systems**, Springer-Verlag, 1996.
- Prasad, S. et al. [2004] "SyD: A Middleware Testbed for Collaborative Applications over Small Heterogeneous Devices and Data Stores," *Proc. ACM/IFIP/USENIX 5th International Middleware Conference (MW-04)*, Toronto, Canada, October 2004.
- Price, B. [2004] "ESRI Systems Integration Technical Brief—ArcSDE High-Availability Overview," ESRI, 2004, Rev 2 ([www.lincoln.ne.gov/city/pworks/gis/pdf/arcsde.pdf](http://www.lincoln.ne.gov/city/pworks/gis/pdf/arcsde.pdf)).
- Rabitti, F., Bertino, E., Kim, W., and Woelk, D. [1991] "A Model of Authorization for Next-Generation Database Systems," *TODS*, 16:1, March 1991.
- Ramakrishnan, R., and Gehrke, J. [2003] **Database Management Systems**, 3rd ed., McGraw-Hill, 2003.
- Ramakrishnan, R., and Ullman, J. [1995] "Survey of Research in Deductive Database Systems," *Journal of Logic Programming*, 23:2, 1995, pp. 125–149.
- Ramakrishnan, R., ed. [1995] **Applications of Logic Databases**, Kluwer Academic, 1995.
- Ramakrishnan, R., Srivastava, D., and Sudarshan, S. [1992] "{CORAL} : {C}ontrol, {R}elations and {L}ogic," in *VLDB* [1992].
- Ramakrishnan, R., Srivastava, D., Sudarshan, S., and Sheshadri, P. [1993] "Implementation of the {CORAL} deductive database system," in *SIGMOD* [1993].
- Ramamoorthy, C., and Wah, B. [1979] "The Placement of Relations on a Distributed Relational Database," *Proc. First International Conference on Distributed Computing Systems*, IEEE CS, 1979.
- Ramesh, V., and Ram, S. [1997] "Integrity Constraint Integration in Heterogeneous Databases an Enhanced Methodology for Schema Integration," *Information Systems*, 22:8, December 1997, pp. 423–446.
- Ratnasamy, S. et al. [2001] "A Scalable Content-Addressable Network," *SIGCOMM* 2001.
- Reed, D. P. [1983] "Implementing Atomic Actions on Decentralized Data," *TOCS*, 1:1, February 1983, pp. 3–23.
- Reese, G. [1997] **Database Programming with JDBC and Java**, O'Reilly, 1997.
- Reisner, P. [1977] "Use of Psychological Experimentation as an Aid to Development of a Query Language," *TSE*, 3:3, May 1977.
- Reisner, P. [1981] "Human Factors Studies of Database Query Languages: A Survey and Assessment," *ACM Computing Surveys*, 13:1, March 1981.
- Reiter, R. [1984] "Towards a Logical Reconstruction of Relational Database Theory," in Brodie et al., Ch. 8 [1984].
- Reuter, A. [1980] "A Fast Transaction Oriented Logging Scheme for UNDO recovery," *TSE* 6:4, pp. 348–356.
- Revilak, S., O'Neil, P., and O'Neil, E. [2011] "Precisely Serializable Snapshot Isolation (PSSI)," in *ICDE* [2011], pp. 482–493.
- Ries, D., and Stonebraker, M. [1977] "Effects of Locking Granularity in a Database Management System," *TODS*, 2:3, September 1977.
- Rissanen, J. [1977] "Independent Components of Relations," *TODS*, 2:4, December 1977.
- Rivest, R. et al. [1978] "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *CACM*, 21:2, February 1978, pp. 120–126.
- Robbins, R. [1993] "Genome Informatics: Requirements and Challenges," *Proc. Second International Conference on Bioinformatics, Supercomputing and Complex Genome Analysis*, World Scientific Publishing, 1993.
- Robertson, S. [1997] "The Probability Ranking Principle in IR," in *Readings in Information Retrieval* (Jones, K. S., and Willett, P., eds.), Morgan Kaufmann Multimedia Information and Systems Series, pp. 281–286.
- Robertson, S., Walker, S., and Hancock-Beaulieu, M. [1995] "Large Test Collection Experiments on an Operational, Interactive System: Okapi at TREC," *Information Processing and Management*, 31, pp. 345–360.
- Rocchio, J. [1971] "Relevance Feedback in Information Retrieval," in *The SMART Retrieval System: Experiments in Automatic Document Processing*, (G. Salton, ed.), Prentice-Hall, pp. 313–323.
- Rosenkrantz, D., Stearns, D., and Lewis, P. [1978] "System-Level Concurrency Control for Distributed Database Systems," *TODS*, 3:2, pp. 178–198.
- Rotem, D., [1991] "Spatial Join Indices," in *ICDE* [1991].
- Roth, M. A., Korth, H. F., and Silberschatz, A. [1988] "Extended Algebra and Calculus for Non-1NF Relational Databases," *TODS*, 13:4, 1988, pp. 389–417.
- Roth, M., and Korth, H. [1987] "The Design of Non-1NF Relational Databases into Nested Normal Form," in *SIGMOD* [1987].
- Rothnie, J. et al. [1980] "Introduction to a System for Distributed Databases (SDD-1)," *TODS*, 5:1, March 1980.



- Roussopoulos, N. [1991] "An Incremental Access Method for View-Cache: Concept, Algorithms, and Cost Analysis," **TODS**, 16:3, September 1991.
- Roussopoulos, N., Kelley, S., and Vincent, F. [1995] "Nearest Neighbor Queries," in *SIGMOD* [1995], pp. 71–79.
- Rozen, S., and Shasha, D. [1991] "A Framework for Automating Physical Database Design," in *VLDB* [1991].
- Rudensteiner, E. [1992] "Multiview: A Methodology for Supporting Multiple Views in Object-Oriented Databases," in *VLDB* [1992].
- Ruemmler, C., and Wilkes, J. [1994] "An Introduction to Disk Drive Modeling," **IEEE Computer**, 27:3, March 1994, pp. 17–27.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. [1991] **Object Oriented Modeling and Design**, Prentice-Hall, 1991.
- Rumbaugh, J., Jacobson, I., Booch, G. [1999] **The Unified Modeling Language Reference Manual**, Addison-Wesley, 1999.
- Rusinkiewicz, M. et al. [1988] "OMNIBASE—A Loosely Coupled: Design and Implementation of a Multidatabase System," **IEEE Distributed Processing Newsletter**, 10:2, November 1988.
- Rustin, R., ed. [1972] **Data Base Systems**, Prentice-Hall, 1972.
- Rustin, R., ed. [1974] *Proc. BJNAV2*.
- Sacca, D., and Zaniolo, C. [1987] "Implementation of Recursive Queries for a Data Language Based on Pure Horn Clauses," *Proc. Fourth International Conference on Logic Programming*, MIT Press, 1986.
- Sadri, F., and Ullman, J. [1982] "Template Dependencies: A Large Class of Dependencies in Relational Databases and Its Complete Axiomatization," **JACM**, 29:2, April 1982.
- Sagiv, Y., and Yannakakis, M. [1981] "Equivalence among Relational Expressions with the Union and Difference Operators," **JACM**, 27:4, November 1981.
- Sahay, S. et al. [2008] "Discovering Semantic Biomedical Relations Utilizing the Web," in **Journal of ACM Transactions on Knowledge Discovery from Data (TKDD)**, Special issue on Bioinformatics, 2:1, 2008.
- Sakai, H. [1980] "Entity-Relationship Approach to Conceptual Schema Design," in *SIGMOD* [1980].
- Salem, K., and Garcia-Molina, H. [1986] "Disk Striping," in *ICDE* [1986], pp. 336–342.
- Salton, G. [1968] **Automatic Information Organization and Retrieval**, McGraw Hill, 1968.
- Salton, G. [1971] **The SMART Retrieval System—Experiments in Automatic Document Processing**, Prentice-Hall, 1971.
- Salton, G. [1990] "Full Text Information Processing Using the Smart System," **IEEE Data Engineering Bulletin** 13:1, 1990, pp. 2–9.
- Salton, G., and Buckley, C. [1991] "Global Text Matching for Information Retrieval" in **Science**, 253, August 1991.
- Salton, G., Yang, C. S., and Yu, C. T. [1975] "A theory of term importance in automatic text analysis," **Journal of the American Society for Information Science**, 26, pp. 33–44 (1975).
- Salzberg, B. [1988] **File Structures: An Analytic Approach**, Prentice-Hall, 1988.
- Salzberg, B. et al. [1990] "FastSort: A Distributed Single-Input Single-Output External Sort," in *SIGMOD* [1990].
- Samet, H. [1990] **The Design and Analysis of Spatial Data Structures**, Addison-Wesley, 1990.
- Samet, H. [1990a] **Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS**, Addison-Wesley, 1990.
- Sammur, C., and Sammur, R. [1983] "The Implementation of UNSW-PROLOG," **The Australian Computer Journal**, May 1983.
- Santucci, G. [1998] "Semantic Schema Refinements for Multilevel Schema Integration," **DKE**, 25:3, 1998, pp. 301–326.
- Sarasua, W., and O'Neill, W. [1999]. "GIS in Transportation," in Taylor and Francis [1999].
- Sarawagi, S., Thomas, S., and Agrawal, R. [1998] "Integrating Association Rules Mining with Relational Database systems: Alternatives and Implications," in *SIGMOD* [1998].
- Savasere, A., Omiecinski, E., and Navathe, S. [1995] "An Efficient Algorithm for Mining Association Rules," in *VLDB* [1995].
- Savasere, A., Omiecinski, E., and Navathe, S. [1998] "Mining for Strong Negative Association in a Large Database of Customer Transactions," in *ICDE* [1998].
- Schatz, B. [1995] "Information Analysis in the Net: The Interspace of the Twenty-First Century," *Keynote Plenary Lecture at American Society for Information Science (ASIS) Annual Meeting*, Chicago, October 11, 1995.
- Schatz, B. [1997] "Information Retrieval in Digital Libraries: Bringing Search to the Net," **Science**, 275:17 January 1997.
- Schek, H. J., and Scholl, M. H. [1986] "The Relational Model with Relation-valued Attributes," **Information Systems**, 11:2, 1986.
- Schek, H. J., Paul, H. B., Scholl, M. H., and Weikum, G. [1990] "The DASDBS Project: Objects, Experiences, and Future Projects," **TKDE**, 2:1, 1990.
- Scheuermann, P., Schiffner, G., and Weber, H. [1979] "Abstraction Capabilities and Invariant Properties Modeling within the Entity-Relationship Approach," in *ER Conference* [1979].
- Schlimmer, J., Mitchell, T., and McDermott, J. [1991] "Justification Based Refinement of Expert Knowledge" in *Piatetsky-Shapiro and Frawley* [1991].

- Schmarzo, B. [2013] **Big Data: Understanding How Data Powers Big Business**, Wiley, 2013, 240 pp.
- Schlossnagle, G. [2005] **Advanced PHP Programming**, Sams, 2005.
- Schmidt, J., and Swenson, J. [1975] "On the Semantics of the Relational Model," in *SIGMOD* [1975].
- Schneider, R. D. [2006] **MySQL Database Design and Tuning**, MySQL Press, 2006.
- Scholl, M. O., Voisard, A., and Rigaux, P. [2001] **Spatial Database Management Systems**, Morgan Kaufman, 2001.
- Sciore, E. [1982] "A Complete Axiomatization for Full Join Dependencies," *JACM*, 29:2, April 1982.
- Scott, M., and Fowler, K. [1997] **UML Distilled: Applying the Standard Object Modeling Language**, Addison-Wesley, 1997.
- Selinger, P. et al. [1979] "Access Path Selection in a Relational Database Management System," in *SIGMOD* [1979].
- Senko, M. [1975] "Specification of Stored Data Structures and Desired Output in DIAM II with FORAL," in *VLDB* [1975].
- Senko, M. [1980] "A Query Maintenance Language for the Data Independent Accessing Model II," **Information Systems**, 5:4, 1980.
- Shapiro, L. [1986] "Join Processing in Database Systems with Large Main Memories," *TODS*, 11:3, 1986.
- Shasha, D., and Bonnet, P. [2002] **Database Tuning: Principles, Experiments, and Troubleshooting Techniques**, Morgan Kaufmann, Revised ed., 2002.
- Shasha, D., and Goodman, N. [1988] "Concurrent Search Structure Algorithms," *TODS*, 13:1, March 1988.
- Shekhar, S., and Chawla, S. [2003] **Spatial Databases, A Tour**, Prentice-Hall, 2003.
- Shekhar, S., and Xiong, H. [2008] **Encyclopedia of GIS**, Springer Link (Online service).
- Shekita, E., and Carey, M. [1989] "Performance Enhancement Through Replication in an Object-Oriented DBMS," in *SIGMOD* [1989].
- Shenoy, S., and Ozsoyoglu, Z. [1989] "Design and Implementation of a Semantic Query Optimizer," *TKDE*, 1:3, September 1989.
- Sheth, A. P., and Larson, J. A. [1990] "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, 22:3, September 1990, pp. 183–236.
- Sheth, A., Gala, S., and Navathe, S. [1993] "On Automatic Reasoning for Schema Integration," in **International Journal of Intelligent Co-operative Information Systems**, 2:1, March 1993.
- Sheth, A., Larson, J., Cornelio, A., and Navathe, S. [1988] "A Tool for Integrating Conceptual Schemas and User Views," in *ICDE* [1988].
- Shipman, D. [1981] "The Functional Data Model and the Data Language DAPLEX," *TODS*, 6:1, March 1981.
- Shlaer, S., Mellor, S. [1988] **Object-Oriented System Analysis: Modeling the World in Data**, Prentice-Hall, 1988.
- Shneiderman, B., ed. [1978] **Databases: Improving Usability and Responsiveness**, Academic Press, 1978.
- Shvachko, K.V. [2012] "HDFS Scalability: the limits of growth," **Usenix legacy publications, Login**, Vol. 35, No. 2, pp. 6–16, April 2010 (<https://www.usenix.org/legacy/publications/login/2010-04/openpdfs/shvachko.pdf>)
- Sibley, E., and Kerschberg, L. [1977] "Data Architecture and Data Model Considerations," *NCC, AFIPS*, 46, 1977.
- Siegel, M., and Madnick, S. [1991] "A Metadata Approach to Resolving Semantic Conflicts," in *VLDB* [1991].
- Siegel, M., Sciore, E., and Salveter, S. [1992] "A Method for Automatic Rule Derivation to Support Semantic Query Optimization," *TODS*, 17:4, December 1992.
- SIGMOD* [1974] *Proc. ACM SIGMOD-SIGFIDET Conference on Data Description, Access, and Control*, Rustin, R., ed., May 1974.
- SIGMOD* [1975] *Proc. 1975 ACM SIGMOD International Conference on Management of Data*, King, F., ed., San Jose, CA, May 1975.
- SIGMOD* [1976] *Proc. 1976 ACM SIGMOD International Conference on Management of Data*, Rothnie, J., ed., Washington, June 1976.
- SIGMOD* [1977] *Proc. 1977 ACM SIGMOD International Conference on Management of Data*, Smith, D., ed., Toronto, August 1977.
- SIGMOD* [1978] *Proc. 1978 ACM SIGMOD International Conference on Management of Data*, Lowenthal, E., and Dale, N., eds., Austin, TX, May/June 1978.
- SIGMOD* [1979] *Proc. 1979 ACM SIGMOD International Conference on Management of Data*, Bernstein, P., ed., Boston, MA, May/June 1979.
- SIGMOD* [1980] *Proc. 1980 ACM SIGMOD International Conference on Management of Data*, Chen, P., and Sprowls, R., eds., Santa Monica, CA, May 1980.
- SIGMOD* [1981] *Proc. 1981 ACM SIGMOD International Conference on Management of Data*, Lien, Y., ed., Ann Arbor, MI, April/May 1981.
- SIGMOD* [1982] *Proc. 1982 ACM SIGMOD International Conference on Management of Data*, Schkolnick, M., ed., Orlando, FL, June 1982.
- SIGMOD* [1983] *Proc. 1983 ACM SIGMOD International Conference on Management of Data*, DeWitt, D., and Gardarin, G., eds., San Jose, CA, May 1983.
- SIGMOD* [1984] *Proc. 1984 ACM SIGMOD International Conference on Management of Data*, Yormark, E., ed., Boston, MA, June 1984.

- SIGMOD [1985] *Proc. 1985 ACM SIGMOD International Conference on Management of Data*, Navathe, S., ed., Austin, TX, May 1985.
- SIGMOD [1986] *Proc. 1986 ACM SIGMOD International Conference on Management of Data*, Zaniolo, C., ed., Washington, May 1986.
- SIGMOD [1987] *Proc. 1987 ACM SIGMOD International Conference on Management of Data*, Dayal, U., and Traiger, I., eds., San Francisco, CA, May 1987.
- SIGMOD [1988] *Proc. 1988 ACM SIGMOD International Conference on Management of Data*, Boral, H., and Larson, P., eds., Chicago, June 1988.
- SIGMOD [1989] *Proc. 1989 ACM SIGMOD International Conference on Management of Data*, Clifford, J., Lindsay, B., and Maier, D., eds., Portland, OR, June 1989.
- SIGMOD [1990] *Proc. 1990 ACM SIGMOD International Conference on Management of Data*, Garcia-Molina, H., and Jagadish, H., eds., Atlantic City, NJ, June 1990.
- SIGMOD [1991] *Proc. 1991 ACM SIGMOD International Conference on Management of Data*, Clifford, J., and King, R., eds., Denver, CO, June 1991.
- SIGMOD [1992] *Proc. 1992 ACM SIGMOD International Conference on Management of Data*, Stonebraker, M., ed., San Diego, CA, June 1992.
- SIGMOD [1993] *Proc. 1993 ACM SIGMOD International Conference on Management of Data*, Buneman, P., and Jajodia, S., eds., Washington, June 1993.
- SIGMOD [1994] *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Snodgrass, R. T., and Winslett, M., eds., Minneapolis, MN, June 1994.
- SIGMOD [1995] *Proceedings of 1995 ACM SIGMOD International Conference on Management of Data*, Carey, M., and Schneider, D. A., eds., Minneapolis, MN, June 1995.
- SIGMOD [1996] *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data*, Jagadish, H. V., and Mumick, I. P., eds., Montreal, June 1996.
- SIGMOD [1997] *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*, Peckham, J., ed., Tucson, AZ, May 1997.
- SIGMOD [1998] *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data*, Haas, L., and Tiwary, A., eds., Seattle, WA, June 1998.
- SIGMOD [1999] *Proceedings of 1999 ACM SIGMOD International Conference on Management of Data*, Faloutsos, C., ed., Philadelphia, PA, May 1999.
- SIGMOD [2000] *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data*, Chen, W., Naughton J., and Bernstein, P., eds., Dallas, TX, May 2000.
- SIGMOD [2001] *Proceedings of 2001 ACM SIGMOD International Conference on Management of Data*, Aref, W., ed., Santa Barbara, CA, May 2001.
- SIGMOD [2002] *Proceedings of 2002 ACM SIGMOD International Conference on Management of Data*, Franklin, M., Moon, B., and Ailamaki, A., eds., Madison, WI, June 2002.
- SIGMOD [2003] *Proceedings of 2003 ACM SIGMOD International Conference on Management of Data*, Halevy, Y., Zachary, G., and Doan, A., eds., San Diego, CA, June 2003.
- SIGMOD [2004] *Proceedings of 2004 ACM SIGMOD International Conference on Management of Data*, Weikum, G., Christian König, A., and DeBloch, S., eds., Paris, France, June 2004.
- SIGMOD [2005] *Proceedings of 2005 ACM SIGMOD International Conference on Management of Data*, Widom, J., ed., Baltimore, MD, June 2005.
- SIGMOD [2006] *Proceedings of 2006 ACM SIGMOD International Conference on Management of Data*, Chaudhari, S., Hristidis, V., and Polyzotis, N., eds., Chicago, IL, June 2006.
- SIGMOD [2007] *Proceedings of 2007 ACM SIGMOD International Conference on Management of Data*, Chan, C.-Y., Ooi, B.-C., and Zhou, A., eds., Beijing, China, June 2007.
- SIGMOD [2008] *Proceedings of 2008 ACM SIGMOD International Conference on Management of Data*, Wang, J. T.-L., ed., Vancouver, Canada, June 2008.
- SIGMOD [2009] *Proceedings of 2009 ACM SIGMOD International Conference on Management of Data*, Cetintemel, U., Zdonik, S., Kossman, D., and Tatbul, N., eds., Providence, RI, June–July 2009.
- SIGMOD [2010] *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*, Elmagarmid, Ahmed K. and Agrawal, Divyakant eds., Indianapolis, IN, June 2010.
- SIGMOD [2011] *Proceedings of 2011 ACM SIGMOD International Conference on Management of Data*, Sellis, T., Miller, R., Kementsietsidis, A., and Velegrakis, Y., eds., Athens, Greece, June 2011.
- SIGMOD [2012] *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*, Selcuk Candan, K., Chen, Yi, Snodgrass, R., Gravano, L., Fuxman, A., eds., Scottsdale, Arizona, June 2012.
- SIGMOD [2013] *Proceedings of 2013 ACM SIGMOD International Conference on Management of Data*, Ross, K., Srivastava, D., Papadias, D., eds, New York, June 2013.
- SIGMOD [2014] *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data*, Dyreson, C., Li, Feifei., Ozsu, T., eds., Snowbird, UT, June 2014.
- SIGMOD [2015] *Proceedings of 2015 ACM SIGMOD International Conference on Management of Data*, Melbourne, Australia, May-June 2015, forthcoming.
- Silberschatz, A., Korth, H., and Sudarshan, S. [2011] **Database System Concepts**, 6th ed., McGraw-Hill, 2011.

- Silberschatz, A., Stonebraker, M., and Ullman, J. [1990] "Database Systems: Achievements and Opportunities," in **ACM SIGMOD Record**, 19:4, December 1990.
- Simon, H. A. [1971] "Designing Organizations for an Information-Rich World," in **Computers, Communications and the Public Interest**, (Greenberger, M., ed.), The Johns Hopkins University Press, 1971, (pp. 37–72).
- Sion, R., Atallah, M., and Prabhakar, S. [2004] "Protecting Rights Proofs for Relational Data Using Watermarking," **TKDE**, 16:12, 2004, pp. 1509–1525.
- Sklar, D. [2005] **Learning PHP5**, O'Reilly Media, Inc., 2005.
- Smith, G. [1990] "The Semantic Data Model for Security: Representing the Security Semantics of an Application," in **ICDE** [1990].
- Smith, J. et al. [1981] "MULTIBASE: Integrating Distributed Heterogeneous Database Systems," **NCC, AFIPS**, 50, 1981.
- Smith, J. R., and Chang, S.-F. [1996] "VisualSEEK: A Fully Automated Content-Based Image Query System," *Proc. 4th ACM Multimedia Conf.*, Boston, MA, November 1996, pp. 87–98.
- Smith, J., and Chang, P. [1975] "Optimizing the Performance of a Relational Algebra Interface," **CACM**, 18:10, October 1975.
- Smith, J., and Smith, D. [1977] "Database Abstractions: Aggregation and Generalization," **TODS**, 2:2, June 1977.
- Smith, K., and Winslett, M. [1992] "Entity Modeling in the MLS Relational Model," in **VLDB** [1992].
- Smith, P., and Barnes, G. [1987] **Files and Databases: An Introduction**, Addison-Wesley, 1987.
- Snodgrass, R. [1987] "The Temporal Query Language TQuel," **TODS**, 12:2, June 1987.
- Snodgrass, R., and Ahn, I. [1985] "A Taxonomy of Time in Databases," in **SIGMOD** [1985].
- Snodgrass, R., ed. [1995] **The TSQL2 Temporal Query Language**, Springer, 1995.
- Soutou, G. [1998] "Analysis of Constraints for N-ary Relationships," in **ER98**.
- Spaccapietra, S., and Jain, R., eds. [1995] *Proc. Visual Database Workshop*, Lausanne, Switzerland, October 1995.
- Spiliopoulou, M. [2000] "Web Usage Mining for Web Site Evaluation," **CACM** 43:8, August 2000, pp. 127–134.
- Spooner D., Michael, A., and Donald, B. [1986] "Modeling CAD Data with Data Abstraction and Object-Oriented Technique," in **ICDE** [1986].
- Srikant, R., and Agrawal, R. [1995] "Mining Generalized Association Rules," in **VLDB** [1995].
- Srinivas, M., and Patnaik, L. [1994] "Genetic Algorithms: A Survey," **IEEE Computer**, 27:6, June 1994, pp.17–26.
- Srinivasan, V., and Carey, M. [1991] "Performance of B-Tree Concurrency Control Algorithms," in **SIGMOD** [1991].
- Srivastava, D., Ramakrishnan, R., Sudarshan, S., and Sheshadri, P. [1993] "Coral++: Adding Object-orientation to a Logic Database Language," in **VLDB** [1993].
- Srivastava, J. et al. [2000] "Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data," **SIGKDD Explorations**, 1:2, 2000.
- Stachour, P., and Thuraisingham, B. [1990] "The Design and Implementation of INGRES," **TKDE**, 2:2, June 1990.
- Stallings, W. [1997] **Data and Computer Communications**, 5th ed., Prentice-Hall, 1997.
- Stallings, W. [2010] **Network Security Essentials, Applications and Standards**, 4th ed., Prentice-Hall, 2010.
- Stevens, P., and Pooley, R. [2003] **Using UML: Software Engineering with Objects and Components**, Revised edition, Addison-Wesley, 2003.
- Stoesser, G. et al. [2003] "The EMBL Nucleotide Sequence Database: Major New Developments," **Nucleic Acids Research**, 31:1, January 2003, pp. 17–22.
- Stoica, I., Morris, R., Karger, D. et al. [2001] "Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications," **SIGCOMM** 2001.
- Stonebraker, M., Aoki, P., Litwin W., et al. [1996] "Mariposa: A Wide-Area Distributed Database System" **VLDB J**, 5:1, 1996, pp. 48–63.
- Stonebraker M. et al. [2005] "C-store: A column oriented DBMS," in **VLDB** [2005].
- Stonebraker, M. [1975] "Implementation of Integrity Constraints and Views by Query Modification," in **SIGMOD** [1975].
- Stonebraker, M. [1993] "The Miro DBMS" in **SIGMOD** [1993].
- Stonebraker, M., and Rowe, L. [1986] "The Design of POSTGRES," in **SIGMOD** [1986].
- Stonebraker, M., ed. [1994] **Readings in Database Systems**, 2nd ed., Morgan Kaufmann, 1994.
- Stonebraker, M., Hanson, E., and Hong, C. [1987] "The Design of the POSTGRES Rules System," in **ICDE** [1987].
- Stonebraker, M., with Moore, D. [1996] **Object-Relational DBMSs: The Next Great Wave**, Morgan Kaufmann, 1996.
- Stonebraker, M., Wong, E., Kreps, P., and Held, G. [1976] "The Design and Implementation of INGRES," **TODS**, 1:3, September 1976.
- Stroustrup, B. [1997] **The C++ Programming Language: Special Edition**, Pearson, 1997.
- Su, S. [1985] "A Semantic Association Model for Corporate and Scientific-Statistical Databases," **Information Science**, 29, 1985.
- Su, S. [1988] **Database Computers**, McGraw-Hill, 1988.
- Su, S., Krishnamurthy, V., and Lam, H. [1988] "An Object-Oriented Semantic Association Model (OSAM\*)," in



- AI in Industrial Engineering and Manufacturing: Theoretical Issues and Applications**, American Institute of Industrial Engineers, 1988.
- Subrahmanian V. S., and Jajodia, S., eds. [1996] **Multimedia Database Systems: Issues and Research Directions**, Springer-Verlag, 1996.
- Subrahmanian, V. [1998] **Principles of Multimedia Databases Systems**, Morgan Kaufmann, 1998.
- Sunderraman, R. [2007] **ORACLE 10g Programming: A Primer**, Addison-Wesley, 2007.
- Swami, A., and Gupta, A. [1989] "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques," in *SIGMOD* [1989].
- Sybase [2005] **System Administration Guide: Volume 1 and Volume 2 (Adaptive Server Enterprise 15.0)**, Sybase, 2005.
- Tan, P., Steinbach, M., and Kumar, V. [2006] **Introduction to Data Mining**, Addison-Wesley, 2006.
- Tanenbaum, A. [2003] **Computer Networks**, 4th ed., Prentice-Hall PTR, 2003.
- Tansel, A. et al., eds. [1993] **Temporal Databases: Theory, Design, and Implementation**, Benjamin Cummings, 1993.
- Teorey, T. [1994] **Database Modeling and Design: The Fundamental Principles**, 2nd ed., Morgan Kaufmann, 1994.
- Teorey, T., Yang, D., and Fry, J. [1986] "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," *ACM Computing Surveys*, 18:2, June 1986.
- Thomas, J., and Gould, J. [1975] "A Psychological Study of Query by Example," *NCC AFIPS*, 44, 1975.
- Thomas, R. [1979] "A Majority Consensus Approach to Concurrency Control for Multiple Copy Data Bases," *TODS*, 4:2, June 1979.
- Thomasian, A. [1991] "Performance Limits of Two-Phase Locking," in *ICDE* [1991].
- Thuraisingham, B. [2001] **Managing and Mining Multimedia Databases**, CRC Press, 2001.
- Thuraisingham, B., Clifton, C., Gupta, A., Bertino, E., and Ferrari, E. [2001] "Directions for Web and E-commerce Applications Security," *Proc. 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2001, pp. 200–204.
- Thusoo, A. et al. [2010] Hive—A Petabyte Scale Data Warehouse Using Hadoop, in *ICDE* [2010].
- Todd, S. [1976] "The Peterlee Relational Test Vehicle—A System Overview," *IBM Systems Journal*, 15:4, December 1976.
- Toivonen, H., "Sampling Large Databases for Association Rules," in *VLDB* [1996].
- Tou, J., ed. [1984] **Information Systems COINS-IV**, Plenum Press, 1984.
- Tsangaris, M., and Naughton, J. [1992] "On the Performance of Object Clustering Techniques," in *SIGMOD* [1992].
- Tsichritzis, D. [1982] "Forms Management," *CACM*, 25:7, July 1982.
- Tsichritzis, D., and Klug, A., eds. [1978] **The ANSI/X3/SPARC DBMS Framework**, AFIPS Press, 1978.
- Tsichritzis, D., and Lochovsky, F. [1976] "Hierarchical Database Management: A Survey," *ACM Computing Surveys*, 8:1, March 1976.
- Tsichritzis, D., and Lochovsky, F. [1982] **Data Models**, Prentice-Hall, 1982.
- Tsotras, V., and Gopinath, B. [1992] "Optimal Versioning of Object Classes," in *ICDE* [1992].
- Tsou, D. M., and Fischer, P. C. [1982] "Decomposition of a Relation Scheme into Boyce Codd Normal Form," *SIGACT News*, 14:3, 1982, pp. 23–29.
- U.S. Congress [1988] "Office of Technology Report, Appendix D: Databases, Repositories, and Informatics," in **Mapping Our Genes: Genome Projects: How Big, How Fast?** John Hopkins University Press, 1988.
- U.S. Department of Commerce [1993] **TIGER/Line Files**, Bureau of Census, Washington, 1993.
- Ullman, J. [1982] **Principles of Database Systems**, 2nd ed., Computer Science Press, 1982.
- Ullman, J. [1985] "Implementation of Logical Query Languages for Databases," *TODS*, 10:3, September 1985.
- Ullman, J. [1988] **Principles of Database and Knowledge-Base Systems**, Vol. 1, Computer Science Press, 1988.
- Ullman, J. [1989] **Principles of Database and Knowledge-Base Systems**, Vol. 2, Computer Science Press, 1989.
- Ullman, J. D., and Widom, J. [1997] **A First Course in Database Systems**, Prentice-Hall, 1997.
- Uschold, M., and Gruninger, M. [1996] "Ontologies: Principles, Methods and Applications," *Knowledge Engineering Review*, 11:2, June 1996.
- Vadivelu, V., Jayakumar, R. V., Muthuvel, M., et al. [2008] "A backup mechanism with concurrency control for multilevel secure distributed database systems," *Proc. Int. Conf. on Digital Information Management*, 2008, pp. 57–62.
- Vaidya, J., and Clifton, C., "Privacy-Preserving Data Mining: Why, How, and What For?" *IEEE Security & Privacy (IEEE SP)*, November–December 2004, pp. 19–27.
- Valduriez, P., and Gardarin, G. [1989] **Analysis and Comparison of Relational Database Systems**, Addison-Wesley, 1989.
- van Rijsbergen, C. J. [1979] **Information Retrieval**, Butterworths, 1979.
- Valiant, L. [1990] "A Bridging Model for Parallel Computation," *CACM*, 33:8, August 1990.
- Vassiliou, Y. [1980] "Functional Dependencies and Incomplete Information," in *VLDB* [1980].

- Vélez, F., Bernard, G., Darnis, V. [1989] "The O2 Object Manager: an Overview." In *VLDB* [1989], pp. 357–366.
- Verheijen, G., and VanBekkum, J. [1982] "NIAM: An Information Analysis Method," in Olle et al. [1982].
- Verhofstad, J. [1978] "Recovery Techniques for Database Systems," *ACM Computing Surveys*, 10:2, June 1978.
- Vielle, L. [1986] "Recursive Axioms in Deductive Databases: The Query-Subquery Approach," in *EDS* [1986].
- Vielle, L. [1987] "Database Complete Proof Production Based on SLD-resolution," in *Proc. Fourth International Conference on Logic Programming*, 1987.
- Vielle, L. [1988] "From QSQ Towards QoSQ: Global Optimization of Recursive Queries," in *EDS* [1988].
- Vielle, L. [1998] "VALIDITY: Knowledge Independence for Electronic Mediation," invited paper, in *Practical Applications of Prolog/Practical Applications of Constraint Technology (PAP/PACT '98)*, London, March 1998.
- Vin, H., Zellweger, P., Swinehart, D., and Venkat Rangan, P. [1991] "Multimedia Conferencing in the Etherphone Environment," *IEEE Computer*, Special Issue on Multimedia Information Systems, 24:10, October 1991.
- VLDB* [1975] *Proc. First International Conference on Very Large Data Bases*, Kerr, D., ed., Framingham, MA, September 1975.
- VLDB* [1976] *Systems for Large Databases*, Lockemann, P., and Neuhold, E., eds., in *Proc. Second International Conference on Very Large Data Bases*, Brussels, Belgium, July 1976, North-Holland, 1976.
- VLDB* [1977] *Proc. Third International Conference on Very Large Data Bases*, Merten, A., ed., Tokyo, Japan, October 1977.
- VLDB* [1978] *Proc. Fourth International Conference on Very Large Data Bases*, Bubenko, J., and Yao, S., eds., West Berlin, Germany, September 1978.
- VLDB* [1979] *Proc. Fifth International Conference on Very Large Data Bases*, Furtado, A., and Morgan, H., eds., Rio de Janeiro, Brazil, October 1979.
- VLDB* [1980] *Proc. Sixth International Conference on Very Large Data Bases*, Lochovsky, F., and Taylor, R., eds., Montreal, Canada, October 1980.
- VLDB* [1981] *Proc. Seventh International Conference on Very Large Data Bases*, Zaniolo, C., and Delobel, C., eds., Cannes, France, September 1981.
- VLDB* [1982] *Proc. Eighth International Conference on Very Large Data Bases*, McLeod, D., and Villasenor, Y., eds., Mexico City, September 1982.
- VLDB* [1983] *Proc. Ninth International Conference on Very Large Data Bases*, Schkolnick, M., and Thanos, C., eds., Florence, Italy, October/November 1983.
- VLDB* [1984] *Proc. Tenth International Conference on Very Large Data Bases*, Dayal, U., Schlageter, G., and Seng, L., eds., Singapore, August 1984.
- VLDB* [1985] *Proc. Eleventh International Conference on Very Large Data Bases*, Pirotte, A., and Vassiliou, Y., eds., Stockholm, Sweden, August 1985.
- VLDB* [1986] *Proc. Twelfth International Conference on Very Large Data Bases*, Chu, W., Gardarin, G., and Ohsuga, S., eds., Kyoto, Japan, August 1986.
- VLDB* [1987] *Proc. Thirteenth International Conference on Very Large Data Bases*, Stocker, P., Kent, W., and Hammersley, P., eds., Brighton, England, September 1987.
- VLDB* [1988] *Proc. Fourteenth International Conference on Very Large Data Bases*, Bancilhon, F., and DeWitt, D., eds., Los Angeles, August/September 1988.
- VLDB* [1989] *Proc. Fifteenth International Conference on Very Large Data Bases*, Apers, P., and Wiederhold, G., eds., Amsterdam, August 1989.
- VLDB* [1990] *Proc. Sixteenth International Conference on Very Large Data Bases*, McLeod, D., Sacks-Davis, R., and Schek, H., eds., Brisbane, Australia, August 1990.
- VLDB* [1991] *Proc. Seventeenth International Conference on Very Large Data Bases*, Lohman, G., Sernadas, A., and Camps, R., eds., Barcelona, Catalonia, Spain, September 1991.
- VLDB* [1992] *Proc. Eighteenth International Conference on Very Large Data Bases*, Yuan, L., ed., Vancouver, Canada, August 1992.
- VLDB* [1993] *Proc. Nineteenth International Conference on Very Large Data Bases*, Agrawal, R., Baker, S., and Bell, D. A., eds., Dublin, Ireland, August 1993.
- VLDB* [1994] *Proc. 20th International Conference on Very Large Data Bases*, Bocca, J., Jarke, M., and Zaniolo, C., eds., Santiago, Chile, September 1994.
- VLDB* [1995] *Proc. 21st International Conference on Very Large Data Bases*, Dayal, U., Gray, P.M.D., and Nishio, S., eds., Zurich, Switzerland, September 1995.
- VLDB* [1996] *Proc. 22nd International Conference on Very Large Data Bases*, Vijayaraman, T. M., Buchman, A. P., Mohan, C., and Sarda, N. L., eds., Bombay, India, September 1996.
- VLDB* [1997] *Proc. 23rd International Conference on Very Large Data Bases*, Jarke, M., Carey, M. J., Dittrich, K. R., Lochovsky, F. H., and Loucopoulos, P., eds., Zurich, Switzerland, September 1997.
- VLDB* [1998] *Proc. 24th International Conference on Very Large Data Bases*, Gupta, A., Shmueli, O., and Widom, J., eds., New York, September 1998.
- VLDB* [1999] *Proc. 25th International Conference on Very Large Data Bases*, Zdonik, S. B., Valduriez, P., and Orlowska, M., eds., Edinburgh, Scotland, September 1999.
- VLDB* [2000] *Proc. 26th International Conference on Very Large Data Bases*, Abbadi, A. et al., eds., Cairo, Egypt, September 2000.

- VLDB [2001] *Proc. 27th International Conference on Very Large Data Bases*, Apers, P. et al., eds., Rome, Italy, September 2001.
- VLDB [2002] *Proc. 28th International Conference on Very Large Data Bases*, Bernstein, P., Ionnidis, Y., Ramakrishnan, R., eds., Hong Kong, China, August 2002.
- VLDB [2003] *Proc. 29th International Conference on Very Large Data Bases*, Freytag, J. et al., eds., Berlin, Germany, September 2003.
- VLDB [2004] *Proc. 30th International Conference on Very Large Data Bases*, Nascimento, M. et al., eds., Toronto, Canada, September 2004.
- VLDB [2005] *Proc. 31st International Conference on Very Large Data Bases*, Böhm, K. et al., eds., Trondheim, Norway, August-September 2005.
- VLDB [2006] *Proc. 32nd International Conference on Very Large Data Bases*, Dayal, U. et al., eds., Seoul, Korea, September 2006.
- VLDB [2007] *Proc. 33rd International Conference on Very Large Data Bases*, Koch, C. et al., eds., Vienna, Austria, September, 2007.
- VLDB [2008] *Proc. 34th International Conference on Very Large Data Bases*, as **Proceedings of the VLDB Endowment**, Volume 1, Auckland, New Zealand, August 2008.
- VLDB [2009] *Proc. 35th International Conference on Very Large Data Bases*, as **Proceedings of the VLDB Endowment**, Volume 2, Lyon, France, August 2009.
- VLDB [2010] *Proc. 36th International Conference on Very Large Data Bases*, as **Proceedings of the VLDB Endowment**, Volume 3, Singapore, August 2010.
- VLDB [2011] *Proc. 37th International Conference on Very Large Data Bases*, as **Proceedings of the VLDB Endowment**, Volume 4, Seattle, August 2011.
- VLDB [2012] *Proc. 38th International Conference on Very Large Data Bases*, as **Proceedings of the VLDB Endowment**, Volume 5, Istanbul, Turkey, August 2012.
- VLDB [2013] *Proc. 39th International Conference on Very Large Data Bases*, as **Proceedings of the VLDB Endowment**, Volume 6, Riva del Garda, Trento, Italy, August 2013.
- VLDB [2014] *Proc. 39th International Conference on Very Large Data Bases*, as **Proceedings of the VLDB Endowment**, Volume 7, Hangzhou, China, September 2014.
- VLDB [2015] *Proc. 40th International Conference on Very Large Data Bases*, as **Proceedings of the VLDB Endowment**, Volume 8, Kohala Coast, Hawaii, September 2015, forthcoming.
- Voorhees, E., and Harman, D., eds., [2005] **TREC Experiment and Evaluation in Information Retrieval**, MIT Press, 2005.
- Vorhaus, A., and Mills, R. [1967] "The Time-Shared Data Management System: A New Approach to Data Management," System Development Corporation, Report SP-2634, 1967.
- Wallace, D. [1995] "1994 William Allan Award Address: Mitochondrial DNA Variation in Human Evolution, Degenerative Disease, and Aging," **American Journal of Human Genetics**, 57:201–223, 1995.
- Walton, C., Dale, A., and Jenevein, R. [1991] "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins," in VLDB [1991].
- Wang, K. [1990] "Polynomial Time Designs Toward Both BCNF and Efficient Data Manipulation," in SIGMOD [1990].
- Wang, Y., and Madnick, S. [1989] "The Inter-Database Instance Identity Problem in Integrating Autonomous Systems," in ICDE [1989].
- Wang, Y., and Rowe, L. [1991] "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture," in SIGMOD [1991].
- Warren, D. [1992] "Memoing for Logic Programs," **CACM**, 35:3, ACM, March 1992.
- Weddell, G. [1992] "Reasoning About Functional Dependencies Generalized for Semantic Data Models," **TODS**, 17:1, March 1992.
- Weikum, G. [1991] "Principles and Realization Strategies of Multilevel Transaction Management," **TODS**, 16:1, March 1991.
- Weiss, S., and Indurkha, N. [1998] **Predictive Data Mining: A Practical Guide**, Morgan Kaufmann, 1998.
- Whang, K. [1985] "Query Optimization in Office By Example," IBM Research Report RC 11571, December 1985.
- Whang, K., and Navathe, S. [1987] "An Extended Disjunctive Normal Form Approach for Processing Recursive Logic Queries in Loosely Coupled Environments," in VLDB [1987].
- Whang, K., and Navathe, S. [1992] "Integrating Expert Systems with Database Management Systems—an Extended Disjunctive Normal Form Approach," **Information Sciences**, 64, March 1992.
- Whang, K., Malhotra, A., Sockut, G., and Burns, L. [1990] "Supporting Universal Quantification in a Two-Dimensional Database Query Language," in ICDE [1990].
- Whang, K., Wiederhold, G., and Sagalowicz, D. [1982] "Physical Design of Network Model Databases Using the Property of Separability," in VLDB [1982].
- White, Tom [2012] **Hadoop: The Definitive Guide**, (3rd Ed.), O'Reilly, Yahoo! Press, 2012. [hadoopbook.com].
- Widom, J., "Research Problems in Data Warehousing," CIKM, November 1995.
- Widom, J., and Ceri, S. [1996] **Active Database Systems**, Morgan Kaufmann, 1996.



- Widom, J., and Finkelstein, S. [1990] "Set Oriented Production Rules in Relational Database Systems," in *SIGMOD* [1990].
- Wiederhold, G. [1984] "Knowledge and Database Management," *IEEE Software*, January 1984.
- Wiederhold, G. [1987] **File Organization for Database Design**, McGraw-Hill, 1987.
- Wiederhold, G. [1995] "Digital Libraries, Value, and Productivity," *CACM*, April 1995.
- Wiederhold, G., and Elmasri, R. [1979] "The Structural Model for Database Design," in ER Conference [1979].
- Wiederhold, G., Beetem, A., and Short, G. [1982] "A Database Approach to Communication in VLSI Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1:2, April 1982.
- Wilkinson, K., Lyngbaek, P., and Hasan, W. [1990] "The IRIS Architecture and Implementation," *TKDE*, 2:1, March 1990.
- Willshire, M. [1991] "How Spacey Can They Get? Space Overhead for Storage and Indexing with Object-Oriented Databases," in *ICDE* [1991].
- Wilson, B., and Navathe, S. [1986] "An Analytical Framework for Limited Redesign of Distributed Databases," *Proc. Sixth Advanced Database Symposium*, Tokyo, August 1986.
- Wiorkowski, G., and Kull, D. [1992] **DB2: Design and Development Guide**, 3rd ed., Addison-Wesley, 1992.
- Witkowski, A., et al, "Spreadsheets in RDBMS for OLAP," in *SIGMOD* [2003].
- Wirth, N. [1985] **Algorithms and Data Structures**, Prentice-Hall, 1985.
- Witten, I. H., Bell, T. C., and Moffat, A. [1994] **Managing Gigabytes: Compressing and Indexing Documents and Images**, Wiley, 1994.
- Wolfson, O. Chamberlain, S., Kalpakis, K., and Yesha, Y. [2001] "Modeling Moving Objects for Location Based Services," NSF Workshop on Infrastructure for Mobile and Wireless Systems, in *LNCS* 2538, pp. 46–58.
- Wong, E. [1983] "Dynamic Rematerialization: Processing Distributed Queries Using Redundant Data," *TSE*, 9:3, May 1983.
- Wong, E., and Youssefi, K. [1976] "Decomposition—A Strategy for Query Processing," *TODS*, 1:3, September 1976.
- Wong, H. [1984] "Micro and Macro Statistical/Scientific Database Management," in *ICDE* [1984].
- Wood, J., and Silver, D. [1989] **Joint Application Design: How to Design Quality Systems in 40% Less Time**, Wiley, 1989.
- Worboys, M., Duckham, M. [2004] **GIS – A Computing Perspective**, 2nd ed., CRC Press, 2004.
- Wright, A., Carothers, A., and Campbell, H. [2002]. "Gene-environment interactions the BioBank UK study," *Pharmacogenomics Journal*, 2002, pp. 75–82.
- Wu, X., and Ichikawa, T. [1992] "KDA: A Knowledge-based Database Assistant with a Query Guiding Facility," *TKDE* 4:5, October 1992.
- www.oracle.com/ocom/groups/public/@ocompublic/documents/webcontent/039544.pdf.
- Xie, I. [2008] **Interactive Information Retrieval in Digital Environments**, IGI Publishing, Hershey, PA, 2008.
- Xie, W. [2005] "Supporting Distributed Transaction Processing Over Mobile and Heterogeneous Platforms," Ph.D. dissertation, Georgia Tech, 2005.
- Xie, W., Navathe, S., Prasad, S. [2003] "Supporting QoS-Aware Transaction in the Middleware for a System of Mobile Devices (SyD)," in Proc. 1st Int. Workshop on Mobile Distributed Computing in ICDCS '03, Providence, RI, May 2003.
- XML (2005): www.w3.org/XML/.
- Yan, W.P., and Larson, P.A. [1995] "Eager aggregation and Lazy Aggregation," in *VLDB* [1995].
- Yannakakis, Y. [1984] "Serializability by Locking," *JACM*, 31:2, 1984.
- Yao, S. [1979] "Optimization of Query Evaluation Algorithms," *TODS*, 4:2, June 1979.
- Yao, S., ed. [1985] **Principles of Database Design, Vol. 1: Logical Organizations**, Prentice-Hall, 1985.
- Yee, K.-P. et al. [2003] "Faceted metadata for image search and browsing," *Proc. ACM CHI 2003 (Conference on Human Factors in Computing Systems)*, Ft. Lauderdale, FL, pp. 401–408.
- Yee, W. et al. [2002] "Efficient Data Allocation over Multiple Channels at Broadcast Servers," *IEEE Transactions on Computers*, Special Issue on Mobility and Databases, 51:10, 2002.
- Yee, W., Donahoo, M., and Navathe, S. [2001] "Scaling Replica Maintenance in Intermittently Synchronized Databases," in *CIKM*, 2001.
- Yoshitaka, A., and Ichikawa, K. [1999] "A Survey on Content-Based Retrieval for Multimedia Databases," *TKDE*, 11:1, January 1999.
- Youssefi, K. and Wong, E. [1979] "Query Processing in a Relational Database Management System," in *VLDB* [1979].
- Zadeh, L. [1983] "The Role of Fuzzy Logic in the Management of Uncertainty in Expert Systems," in **Fuzzy Sets and Systems**, 11, North-Holland, 1983.
- Zaharia M. et al. [2012] "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in Proc. Usenix Symp. on Networked System Design and Implementation (NSDI) April 2012, pp. 15–28.
- Zaniolo, C. [1976] "Analysis and Design of Relational Schemata for Database Systems," Ph.D. dissertation, University of California, Los Angeles, 1976.

- Zaniolo, C. [1988] "Design and Implementation of a Logic Based Language for Data Intensive Applications," ICLP/SLP 1988, pp. 1666–1687.
- Zaniolo, C. [1990] "Deductive Databases: Theory meets Practice," in EDBT, 1990, pp. 1–15.
- Zaniolo, C. et al. [1986] "Object-Oriented Database Systems and Knowledge Systems," in *EDS* [1984].
- Zaniolo, C. et al. [1997] **Advanced Database Systems**, Morgan Kaufmann, 1997.
- Zantinge, D., and Adriaans, P. [1996] *Managing Client Server*, Addison-Wesley, 1996.
- Zave, P. [1997] "Classification of Research Efforts in Requirements Engineering," **ACM Computing Surveys**, 29:4, December 1997.
- Zeiler, Michael. [1999] **Modeling Our World—The ESRI Guide to Geodatabase Design**, 1999.
- Zhang, T., Ramakrishnan, R., and Livny, M. [1996] "Birch: An Efficient Data Clustering Method for Very Large Databases," in *SIGMOD* [1996].
- Zhao, R., and Grosky, W. [2002] "Bridging the Semantic Gap in Image Retrieval," in **Distributed Multimedia Databases: Techniques and Applications** (Shih, T. K., ed.), Idea Publishing, 2002.
- Zhou, X., and Pu, P. [2002] "Visual and Multimedia Information Management," *Proc. Sixth Working Conf. on Visual Database Systems*, Zhou, X., and Pu, P. (eds.), Brisbane Australia, IFIP Conference Proceedings 216, Kluwer, 2002.
- Ziauddin, M. et al. [2008] "Optimizer Plan Change Management: Improved Stability and Performance in Oracle 11g," in *VLDB* [2008].
- Zicari, R. [1991] "A Framework for Schema Updates in an Object-Oriented Database System," in *ICDE* [1991].
- Zloof, M. [1975] "Query by Example," *NCC, AFIPS*, 44, 1975.
- Zloof, M. [1982] "Office By Example: A Business Language That Unifies Data, Word Processing, and Electronic Mail," **IBM Systems Journal**, 21:3, 1982.
- Zobel, J., Moffat, A., and Sacks-Davis, R. [1992] "An Efficient Indexing Technique for Full-Text Database Systems," in *VLDB* [1992].
- Zvieli, A. [1986] "A Fuzzy Relational Calculus," in *EDS* [1986].

# Index

- '', string notation (single quotation), 182, 196, 347–348
- :, multiple inheritance (colon) notation, 393
- @, XPath attribute names, 444
- =, EQUIJOIN comparison operator, 253
- >, dereferencing in SQL, 386
- >, operation arrow notation, 392
- ←, assignment operation, relational algebra, 245
- ρ, RENAME operator, 245–246
- “, operator notation (double quotation), 196, 347–348
- \$, XQuery variable prefix, 445
- %, arbitrary number replacement symbol, SQL, 195–196
- ( ), SQL notation
  - constraint conditions for assertions, 226
  - explicit set of values, 214
  - tuple value comparisons, 210
- ( ), XML DTD element notation, 434
- \*, SQL notation
  - attribute specification and retrieval, 193
  - tuple rows in query results, 218
- \*, XPath elements (wildcard symbol), 444
- \*\_, NATURAL JOIN comparison operator, 253
- / and //, path separators, XML, 443
- /, escape operator, SQL, 196
- [ ], UDT arrays (brackets), 383
- \_, single character replacement symbol, SQL, 195–196
- ||, concatenation operator (double bar), SQL, 182–183
- d, disjointness constraint notation, 114–115
- ∪, set union operation, 120
- ≡, equivalent to symbol, 274
- σ, SELECT operator, 241
- ⇒, implies symbol, 274
- 1NF, *see* First normal form (1NF)
- 2NF, *see* Second normal form (2NF)
- 3NF, *see* Third normal form (3NF)
- 4NF, *see* Fourth normal form (4NF)
- 5NF, *see* Fifth normal form (5NF)
- Abstraction concepts
  - aggregation, 131–133
  - association, 131–132
  - classification, 130
  - identification, 130–131
  - instantiation, 130
  - knowledge representation (KR) and, 129
- Access control
  - content-based, 1142
  - credentials and, 1142
  - defined, 1126
  - Directory Services Markup Language (DSML) and, 1142
  - e-commerce environment and, 1141
  - mandatory access control (MAC), 1121, 1134–1137
  - mobile applications, 1141–1142
  - row-level, 1139–1140
  - Web policies, 1141–1142
  - XML, 1140–1141, 1142
- Access paths
  - data modeling, 34
  - DBMS classification from, 52
- Action, SQL triggers, 227
- Active database systems, 4, 22
- Active database techniques, SQL, 202
- Active databases
  - design issues, 967–972
  - enhanced data models, 963–974
  - event-condition-action (ECA) model, 963–964
  - expert (knowledge-based) systems, 962–963
  - implementation issues, 967–972
  - triggers, 963–967, 973–974
- Active rules
  - applications for, 972–973
  - event-condition-action (ECA) model, 963–964
  - functionality of, 962
  - statement-level rules in STARBURST, 970–972
- Actuator, disk devices, 551
- Acyclic graphs, 52. *See also* Hierarchies
- Adaptive optimization, Oracle, 735
- ADD CONSTRAINT keyword, SQL, 234
- Advanced Encryption Standards (AES), 1150
- After image (AFIM) updating, 816
- Agent-based approach, Web content analysis, 1053–1054
- Aggregate functions
  - asterisk (\*) for tuple rows of query results, 218
  - discarded NULL values, 218
  - grouping and, 216–218, 260–261
- OQL collections and, 413–414
- parallel algorithms, 686
- QBE (Query-by-Example) language, 1175–1177
- query execution and, 709
- SQL query retrieval and, 216–219
- relational algebra for, 260–261
- Aggregate operation implementation, 678–679
- Aggregation
  - semantic modeling process, 131–133
  - UML class diagrams, 87–88
- Algorithms, concurrency control
  - Thomas's write rule, 795
  - timestamp ordering (TO), 793
- Algorithms, data mining
  - apriori algorithm, 1075–1076
  - BIRCH algorithm, 1090
  - FP-growth algorithm, 1077–1080
  - genetic algorithms (GAs), 1093
  - k-means algorithm, 1088–1089
  - partition algorithm, 1081
  - sampling algorithm, 1076–1077
- Algorithms, database recovery
  - ARIES recovery algorithm, 827–831
  - idempotent operations of, 815
  - NO-UNDO/REDO, 815, 821–823
  - UNDO/REDO, 815
- Algorithms, encryption
  - asymmetric key encryption algorithms, 1151
  - RSA public key encryption algorithm, 1152
  - symmetric key algorithms, 1150–1151
- Algorithms, normalization
  - alternative RDB designs, 524–527
  - BCNF schemas, 522–523
  - dependency preservation, 519–522
  - ER-to-relational mapping, 290–296
  - nonadditive (lossless) join property decomposition, 519–523
  - RDB schema design, 519–527
  - 3NF schemas, 519–522
- Algorithms, queries
  - external sorting, 660–663
  - heuristic algebra optimization, 700–701
  - parallel processing, 683–687
  - PROJECT operation, 676–678
  - SELECT operation, 663–668
  - set operation, 676–678
- Alias (tuple variables) of attributes, 192

- ALL option, SQL, 194–195, 210
- All-key relation, 491, 493
- Allocation of file blocks on a disk, 564
- ALTER command, SQL, 233–234
- ALTER TABLE command, SQL, 180
- Analysis, RDB design by, 503
- Analytical data store (ADS), 1105
- Analytical operations, spatial databases, 988
- Anchor texts, 1027
- AND/OR/NOT operators
  - Boolean conditions, 270–271
  - quantifier transformations using, 274
- Annotations, XML language, 440
- Anomalies
  - deletion, 467
  - insertion, 465–466
  - modification, 467
  - RDB design and, 465–467
  - tuple redundant information
    - avoidance using, 465–467
  - update, 465–467
- Anti-join (AJ) operator, 658–660, 677–678, 681, 719–720
- Apache systems
  - Apache Cassandra, 900
  - Apache Giraph, 943
  - Apache Hive, 933–936
  - Apache Pig, 932–933
  - Apache Tez, 943
  - Apache Zookeeper, 900
  - Big data technologies for, 932–936, 943–944
- API (Application programming interface)
  - client-side program calls from, 49
  - data mining, 1095
  - database programming and, 312, 326
  - library of functions, 312, 326
- Application-based (semantic)
  - constraints, 158
- Application development environments, 47
- Application programmers, 16
- Application programs, 6, 313
- Application server, 44, 50
- ApplicationMaster (AM), YARN, 942
- Apriori algorithm, 1075–1076
- Arbitrary number replacement symbol (%), 195–196
- Architecture
  - automated storage tiering (AST), 591
  - centralized DBMS, 46–47
  - client/server, 47–49
  - data independence and, 37–38
  - database systems and, 46–51
  - distributed databases (DDBs), 868–875
  - federated database (FDBS) schema, 871–872
  - Fibre Channel over Ethernet (FCoE), 590–591
  - Fibre Channel over IP (FCIP), 590
  - Internet SCSI (iSCSI), 590
  - label security, 1156–1157
  - mappings, 37
  - network-attached storage (NAS), 589–590
  - n*-tier for Web applications, 49–51
  - parallel database, 683
  - parallel versus distributed, 869
  - pure distributed databases, 869–871
  - shared-disk, 683
  - shared-memory, 683
  - shared-nothing, 684
  - storage area networks (SANs), 588–589
  - storage, 588–592
  - three-schema, 36–38
  - three-tier client/server, 49–51, 872–875
  - two-tier client-server, 49
  - Web applications, 49–51
  - YARN (Hadoop v2), 940–942
- ARIES recovery algorithm, 827–831
- Arithmetic operations, SQL query
  - recovery and, 196–197
- Armstrong's axioms, 506–509
- Array constructor, 369
- Array processing, Oracle, 735–736
- Arrays
  - associative, 350
  - brackets ([]) for, 383
  - dynamic, 345–346
  - numeric, 349
  - PHP programming, 345–346, 348–350
  - UDT elements, 383
- AS option, SQL, 196
- Assertions
  - constraint conditions in parentheses () for, 226
  - CREATE ASSERTION statement, 225–226
  - declarative, 225–227
  - relation schema and, 156
  - SQL constraint specification, 158, 165, 225–226
- Assignment operations ( $\leftarrow$ ), relational algebra, 245
- Association rules
  - apriori algorithm, 1075–1076
  - complications with, 1084
  - confidence of, 1074
  - data mining, 1073–1084
  - FP-growth algorithm, 1077–1080
  - frequent-pattern (FP) tree, 1077–1080
  - hierarchies and, 1081–1082
  - market-basket data model, 1073–1075
  - multidimensional associations, 1082–1083
  - negative associations, 1082–1084
  - partition algorithm, 1081
  - sampling algorithm, 1076–1077
  - support for, 1074
- Association, semantic modeling process, 131–132
- Associations, UML class diagrams 87–88
- Associative arrays, PHP, 350
- Asterisk (\*)
  - all attribute specification, 193
  - tuple rows of query results, 218
- Asymmetric key encryption algorithms, 1151
- Atom constructor, 368, 369
- Atomic (single-valued) types, 368
- Atomic literals, 388
- Atomic objects, ODMG models, 388, 395–398
- Atomic values
  - domains, 151
  - first normal form (1NF), 477–478
  - tuples, 155
- Atomicity property, transactions, 14, 157
- Atoms
  - domain relational calculus formulas, 277–278
  - tuple relational calculus formulas, 270–271
  - truth value of, 270, 277
- Attribute data, 989
- Attribute-defined specialization, 114, 126
- Attribute preservation, RDB
  - decomposition condition, 513
- Attribute versioning, 982–984
- Attributes. *See also* Entities
  - ambiguous, prevention of, 191–192
  - asterisk (\*) for, 193
  - clarity of in RDB design, 461–465
  - complex, 66–67, 441
  - composite, 65–66, 441
  - conceptual data models, 33
  - constraints and defaults in SQL, 184–186
  - data types in SQL, 182–184
  - default values, 184–186
  - defined, 63
  - defining, 114
  - degree (arity) of, 152
  - derived, 66
  - discriminating, 299–300
  - EER-to-relational mapping, 298–300
  - entities and, 63–65
  - ER models, 63–70
  - ER-to-relational mapping, 295–296
  - functional dependency of, 472–473
  - grouping, 219, 260–261
  - HTML tags, 430
  - key (uniqueness constraint), 68–69
  - multiple keys for, 631–632
  - multivalued, 66, 295–296, 481
  - normal form keys, 477
  - NULL values, 66, 184–186
  - ODMG model objects, 396
  - ordered indexes, 631–632
  - partial key, 79
  - prime/nonprime, 477
  - project, 189
  - query retrieval in SQL, 191–192

- RDB design and, 461–465, 472–473
- relation schema and, 152, 461–465
- relational algebra, 245–246
- relational model domains and, 152–153
- relationships as, 74
- relationships types of, 78
- renaming, 192, 214–215, 245–246
- roles for a domain, 152
- semantics for, 461–465
- simple (atomic), 65–66
- single-valued, 66
- SQL use of, 184–186, 191–192
- stored, 66
- subclass specialization, 114
- tree-structured data models, XML, 433
- tuple modification for, 166, 168–169
- update (modify) operation for, 168–169
- value sets (domains) of, 69–70
- versioning, 982–984
- visible/hidden, 371, 375
- XML, 433, 441
- Audio data source analysis, 999
- Audio sources, multimedia databases, 996
- Audit trail, 1127
- Authorization, SQL views as mechanisms of, 232
- AUTHORIZATION command, SQL, 315
- Authorization identifier, SQL schemas, 179
- Automated storage tiering (AST), 591
- Autonomy, DDBs, 845–846
- Auxiliary access structure, 546
- Availability
  - DDBs, 844–845
  - loss of, database threat of, 1122
  - NOSQL, 885–886
- AVERAGE function, grouping, 260
- AVG function, SQL, 217
- Axioms, 1005
- B-trees
  - dynamic multilevel indexes
    - implementation, 617–622
  - file organization and, 583
  - dynamic multilevel index
    - implementation, 617–622
  - physical database design and, 601–602, 617–622
  - unbalanced, 617
  - variations of, 629–630
- B<sup>+</sup>-trees
  - bitmaps for leaf nodes of, 636–637
  - dynamic multilevel index
    - implementation, 622–625
  - physical database design and, 601–602, 622–630
  - search, insert and deletion with, 625–629
  - variations of, 629–630
- Backup and recovery subsystem, 20
- Backup utility, 45
- Bag constructor, 369
- Base class, 127
- Base tables (relations), 180, 182
- Before image (BFIM) updating, 816
- Behavior inheritance, 393
- BETWEEN comparison operator, SQL, 196–197
- Bidirectional associations, UML class diagrams, 87
- Big data storage systems, 3, 26, 31, 51
- Big data technologies
  - Apache systems, 932–936, 943–944
  - cloud computing, 947–949
  - distributed and database combination, 841
  - Hadoop, 916–917, 921–926
  - MapReduce (MR), 917–921, 926–936
  - parallel RDBMS compared to, 944–946
  - technological development of, 911–913
  - variety of data, 915
  - velocity of data, 915
  - veracity of data, 915–916
  - volume of data, 914
  - YARN (Hadoop v2), 936–944, 949–953
- Binary association, UML class diagrams, 87
- Binary locks, 782–784
- Binary operations
  - complete set of, 255
  - DIVISION operation, 255–257
  - JOIN operation, 251–255
  - OUTER JOIN operations, 262–264
  - query tree notation, 257–259
  - relational algebra and, 240, 251–259, 262–264
  - set theory for, 247
- Binary relationships
  - cardinality ratios for, 76–77
  - constraints on, 76–78
  - degree of, 73
  - ER models, 73–74, 76–78
  - ER-to-relational mapping, 293–295
  - existence dependency, 77–78
  - participation constraints, 77–78
  - relationship type, 73–74
  - ternary relationships compared to, 88–91
- Binary search, files, 570
- Bind variables, SQL injection and, 1145–1146
- Binding
  - C++ language binding, 417–418
  - early (static), 344
  - JDBC statement parameters, 333
  - late (dynamic), 377
  - OBDs, 377
  - ODMG standards and, 386, 417–418
  - programming language, 312
  - polymorphism and, 377
  - SQL/CLI statement parameters, 329
- BIRCH algorithm, 1090
- Bitemporal relations, 980–982
- Bit-level striping, RAID, 584, 586
- Bit-string data types, 183
- Bits of data, 547
- Bitmap indexes, 634–637, 1109–1110
- BLOBs (binary large objects, 560–561
- Block-level striping, RAID, 584–585, 586
- Block transfer time, disk devices, 552
- Blocking factor, records, 563
- Blocking records, 563–564
- Boolean data types, 183
- Boolean model, IR, 1030
- Boolean queries, 1035–1036
- Boolean (TRUE/FALSE) statements
  - OQL, 414
  - relational algebra expressions, 241–242
  - SQL query retrieval, 212–214
  - tuple relational calculus formulas, 270–271
- Bottom-tier database server, DBMS as, 344
- Bottom-up conceptual synthesis, 119
- Bottom-up method, RDB design, 460, 504
- Bound columns approach, SQL/CLI query results, 329
- Boyce-Codd normal form (BCNF)
  - decomposition of relations not in, 489–491
  - definition of, 488
  - nonadditive join test for binary decomposition (NJB), 490
  - relations in, 487–489
- Browsing, 1027
- Browsing interfaces, 40
- Bucket join, MapReduce (MR), 931
- Buckets, hashing, 575–576
- Buffer, disk blocks, 550–551
- Buffer replacement policy, 749
- Buffer space, nested-loop join and, 672–673
- Buffering
  - buffer management, 557–558
  - buffer replacement strategies, 559–560
  - CPU processing and, 556–557
  - data using disk devices, 552
  - database recovery, 815–816
  - disk blocks, 541, 556–560, 815–816
  - double buffering technique, 556–557
- Buffering (caching) modules, 20, 42
- Built-in functions, UDT, 384
- Built-in interfaces, ODMG models, 393–396
- Built-in variables, PHP, 352–353
- Bulk loading process, indexes, 639
- Bulk transfer time, disk devices, 552
- Business rules, 21
- Bytes of data, 547
- C language, SQL/CLI (SQI call level interface), 326–331
- C++ language binding, ODMG, 417–418

- Cache memory, 543
- Caching (buffering) disk blocks, database recovery, 815–816
- Calendar, 975
- CALL statement, stored procedures, 337
- Candidate key, 159–160, 477
- Canned transactions, 15
- CAP theorem, NOSQL, 888–890
- Cardinality
  - JOIN operations, 719–720
  - of a relational domain, 152
- CARDINALITY function, 383
- Cardinality ratios, 76–77
- Cartesian product of a relational domain, 153
- CARTESIAN PRODUCT operation, 249–251
- CASCADE option, SQL, 233, 234
- Cascaded values
  - insert violation and, 167
  - SELECT operation sequence of, 243
  - SQL constraint options, 186–187
- Cascading rollback phenomenon
  - database recovery and, 819–821
  - schedules, 762
  - timestamp ordering, 794
- CASE (computer-aided software engineering), 46–47
- CASE clause, SQL, 222–223
- Casual end users, 15–16
- Catalog management, DDBs, 875
- Catalogs
  - component modules and, 42–45
  - DBMS, 10–11, 35, 38, 42–45
  - file storage in, 10–11
  - schema description storage, 35, 38, 180
  - SQL concept, 179–180
- Catastrophic failures, database backup and recovery from, 832–833
- Categories
  - defined, 126
  - EER modeling concept, 108, 120–122, 126
  - EER-to-relational mapping, 302–303
  - partial, 122
  - superclasses and, 120–122
  - total, 122
  - union types using, 120–122, 302–303
- Cautious waiting algorithm, deadlock prevention, 791
- Central processing unit (CPU), primary storage of, 542
- Centralized DBMS, 52
- Centralized DBMS architectures, 46–47
- Certification of transactions, 781
- Certify locks, 796–797
- Chaining, hashing collision resolution, 574
- Character-string data types, 182–183
- Characters of data, 547
- CHECK clauses for, 187
- Checkpoints, database recovery, 818–819, 828–829
- Child nodes, tree structures, 617
- Ciphertext, 1149
- Class diagrams, UML, 85–88
- Class library
  - OOPL (object-oriented programming language) and, 312
  - SQL imported from JDBC, 331, 332
- Classes
  - EER model relationships, 108–110
  - inheritance, 110, 118
  - interface inheritance, ODL, 404–405
  - interfaces, instantiable behavior and, 392
  - Java, 331
  - object data models, 52
  - ODL, 400, 404–405
  - ODMG models, 392, 404–405
  - operations and type definitions, 371
  - property specification, 130
  - subclasses, 108–110, 126
  - superclasses, 109, 110, 126
- Clausal form, deductive databases, 1003–1005
- Client, defined, 48
- Client computer, 44
- Client machines, 47
- Client module, 31
- Client program, 313
- Client/server architectures
  - basic, 47–49
  - centralized DBMS, 46–47
  - two-tier, 49
- Client tier, HTML and, 344
- CLOSE CURSOR command, SQL, 318
- Closed world assumption, 156
- Closure, functional dependencies, 505–506, 508
- Cloud computing
  - Big data technology for, 947–949
  - environment, 31
  - Cloud storage, 3
- Clustered file, 572, 583, 602–603
- Clustering, data mining, 1088–1091
- Clustering indexes, 602, 606–608
- Clusters, file blocks, 564
- Code generator, query processing, 655
- Code injection, SQL, 1144
- Collection (multivalued) constructors, 369
- Collection objects, ODMG models, 393–394
- Collection operators, OQL, 413–416
- Collections
  - built-in interfaces, ODMG, 393–396
  - entity sets, 67–68
  - object extent and, 373, 376
  - persistent, 373, 376
  - transient, 376
- Collision resolution, hashing, 574
- Column, SQL, 179
- Column-based data models, 51, 53
- Column-based NOSQL, 888, 900–903
- Column-based storage of relations, indexing for, 642
- Comments, PHP programming, 345
- Commit point, transaction processing, 756
- Committed projection, schedules, 760
- Communication autonomy, DDBs, 845
- Communication software, DBMS, 46
- Communication variables in embedded SQL, 315, 316
- Commutative property, SELECT operation, 243
- Comparison operators
  - select-from-where query structure and, 188–190
  - select-project-join query structure and, 189, 191
  - SQL query retrieval, 188–191, 195–197
  - substring pattern matching, 195–197
- Compiled queries, 710
- Compilers
  - DBMS interface modules, 42–45
  - DDL for schema definitions, 42–43
  - query, 43–44
  - precompiler, 44
- Complete schedule conditions, 760
- Complete set of relational binary operations, 255
- Completeness (totalness) constraint, 115
- Complex attributes, 66–67
- Complex elements, XML, 431, 441
- Composite, 65–66
- Composite (compound) attributes, XML, 441
- Composite keys, 631
- Concatenation operator (||) in SQL, 182–183
- Concept hierarchy, 1053
- Conceptual (schema) level, 37
- Conceptual data models, 33
- Conceptual design
  - comparison of ODB and RDB, 405–406
  - high-level data model design, 61–62
  - mapping EER schema to ODB schema, 407–408
- Conceptualization, ontology and, 134
- Concurrency
  - control, 749–752, 770–771
  - serializability of schedules and, 770–771
  - transaction processing, 746–747
- Concurrency control protocols, 781
- Concurrency control software, 13–14
- Concurrency control techniques
  - data insertion and, 806
  - deletion operation and, 806
  - distributed databases (DDBs), 854–857
  - granularity of data items, 800–801
  - index concurrency control using locks, 805–806
  - interactive transactions and, 807
  - latches and, 807
  - locking data items, 781



- locks used for, 782–786, 796–797, 805–806
- multiple granularity locking, 801–804
- multiversion concurrency control, 781, 795–797
- phantom records and, 806–807
- snapshot isolation, 781, 799–800
- timestamp ordering (TO), 792–795, 796
- timestamps, 781, 790–791, 793
- two-phase locking (2PL), 782–792, 796–797
- validation (optimistic) of transactions, 781, 798–799
- Conditions
  - constraint parentheses ( ) for
    - assertions, 226
  - trigger component in SQL, 227
- Conflict equivalence, schedules, 765–766
- Conjunctive selection, search methods for, 665–666
- CONNECT TO command, SQL, 315
- Connecting fields for mixed records, 582–583
- Connecting to a database
  - embedded SQL, 315–316
  - PHP, 353–355
- Connection record, SQL/CLI, 327–328
- Connection to database server, 313
- Consistency preservation, transactions, 757
- Constant nodes, query graphs, 273
- Constraint specification language, 165
- Constraints
  - application-based (semantic), 158
  - assertions in SQL, 58, 165, 225–226
  - attribute defaults and, 184–186
  - attribute-defined specialization, 114
  - binary relationships, 76–78
  - business rules, 21
  - CHECK clauses for, 187
  - completeness (totalness), 115
  - conditions in parentheses ( ) for
    - assertions, 226
  - database applications, 21–22, 160–163
  - disjointness (d notation), 114–115
  - domain, 158
  - EER models and, 113–116
  - ER models and, 76–78, 91–92
  - existence dependency, 77–78
  - foreign keys, 163, 186–187
  - generalization, 113–116
  - indexes for management of, 641
  - inherent model-based (implicit), 157
  - inherent rules, 22
  - insert operation and, 166–167
  - integrity, 21–22, 160–163
  - key, 21, 158–160, 163–165, 186–187
  - minimum cardinality, 77
  - naming, 187
  - NULL value and, 160, 163
  - participation, 77–78
  - predicate-defined subclasses, 113–114
  - referential integrity, 21, 186–187
  - relational database schemas, 160–163
  - relational models and, 157–167
  - relationships and, 76–78
  - row-based, 187
  - schema-based (explicit), 157
  - semantics and, 21
  - specialization, 113–116
  - SQL specifications, 165, 184–187, 225–226
  - state, 165
  - structural, 78
  - table-based, 184–187
  - ternary relationships, 91–92
  - transition, 165
  - triggers in SQL, 58, 165
  - UML notation for, 127–128
  - uniqueness, 21
  - user-defined subclasses, 114
  - violations, 166–167
- Constructor function, SQL
  - encapsulation, 384
- Constructors, *see* Type constructors
- Constructs, 35
- Content-based access control, 1142
- Content-based retrieval, 995
- Contiguous allocation, file blocks, 564
- Control measures, database security, 1123–1125
- Conversational information access, IR, 1059
- Conversion of locks, 786
- Core specifications, SQL, 178
- Correlated nested queries, SQL, 211–212
- Cost-based query optimization
  - approach, 710–712
  - defined, 710
  - dynamic programming compared to, 716
  - illustration of, 726–728
- Cost estimation
  - catalog information in cost functions for, 712
  - histograms for, 713
  - JOIN optimization based on cost formulas, 720–721
  - query execution components, 710–712
  - query optimization technique, 657, 710–713, 716–717
  - selection based on cost formulas, 716–717
- Cost functions
  - JOIN operation use of, 717–726
  - query optimization, 714–715, 717–726
  - SELECT operation use of, 714
- COUNT function
  - grouping, 260
  - SQL, 217
- Covert channels, flow control and, 1148–1149
- CREATE ASSERTION statement, SQL, 225–226
- CREATE SCHEMA statement, 179–180
- CREATE TABLE command, SQL, 180–182
- CREATE TRIGGER statement, SQL, 225, 226–227
- CREATE TYPE command, 184, 380–383
- CREATE VIEW statement, SQL, 228–229
- Credentials, access control and, 1142
- CROSS PRODUCT operation
  - relational algebra set theory, 249–251
- SQL tuple combinations, 192–193
- CRUD (create, read, update, and delete) operations, NOSQL, 887, 893, 903
- Cursors
  - declaration of, 317, 319–320
  - impedance mismatch and, 312
  - iterator as, 318
  - SQL query result processing, 312, 317–320
  - updating records, 318
- Cypher query language, Neo4j system, 905–908
- Dangling tuples, RDB design problems, 523–524
- Data
  - Big data technology for, 914–916
  - complex relationships among, 21
  - conceptual representation of, 12
  - databases and, 7–8, 12–14
  - defined, 4
  - directed graph representation of, 427–428
  - elements, 7
  - eXtended Markup Language (XML) and, 25, 426–430
  - granularity of data items, 800–801
  - insulation from programs and, 12–13
  - integrity constraints, 21–22
  - interchanging on the Web, 25
  - logical independence, 37–38
  - multiple views of, 13
  - multiuser transactions and, 13–14
  - physical independence, 38
  - records, 6–7
  - requirements collection and analysis, 60–61
  - self-describing, 10, 427
  - semantics and, 21
  - semistructured, 426–428
  - sharing, 13–14
  - storage, 3–4
  - structured, 426
  - tag notation and use, HTML, 428–430
  - three-schema architecture and, 37–38
  - type, 7–8
  - unstructured, 428–430
  - variety of, 915
  - velocity of, 915
  - veracity of, 915–916
  - volume of, 914
  - virtual, 13



- Data abstraction
  - conceptual representation of, 12–13
  - data models and, 12, 32–34
  - program independence from, 12
- Data allocation, DDBs, 849–853
- Data-based approach, Web content analysis, 1054
- Data buffers, transaction processing, 748–749
- Data-centric documents, XML, 431
- Data collection and records, PHP, 355–356
- Data definition, SQL, 179
- Data dictionary (data repository), 45–46
- Data Encryption Standards (DES), 1150
- Data fragmentation, DDBs, 847–853
- Data independence, three-schema architecture and, 37–38
- Data insertion, concurrency control and, 806
- Data manipulation language (DML), 39–40, 44
- Data marts, 1102
- Data mining
  - application programming interface (API), 1095
  - applications of, 1094
  - association rules, 1073–1084
  - BIRCH algorithm, 1090
  - classification, 1085–1088
  - clustering, 1088–1091
  - commercial tools, 1094–1096
  - data warehousing compared to, 1070
  - decision trees, 1085–1086
  - genetic algorithms (GAs), 1093
  - graphical user interface (GUI), 1095
  - k-means algorithm, 1088–1089
  - knowledge discovery in databases (KDD), 1070–1073
  - neural networks, 1092
  - Open Database Connectivity (ODBC) interface, 1094–1095
  - regression, 1091–1092
  - sequential pattern discovery, 1091
  - spatial databases, 993–994
- Data model mapping
  - database design and, 62
  - logical database design, 289
- Data models. *See also* Object data models
  - access path, 34
  - basic operations, 32
  - categories of, 33–34
  - conceptual, 12–13, 33
  - data abstraction and, 12, 32–34
  - database schemas for, 34–38
  - DBMS classification from, 51–53
  - dynamic aspect of applications, 23
  - EER (enhanced entity-relationship), 107–146
  - ER (entity-relationship), 59–105
  - object, 33, 51, 52–53
  - relational, 33, 51, 52, 149–157
  - representational, 33
  - self-describing, 34
- Data normalization, 475–476
- Data organization transparency, DDBs, 843
- Data quality, database security and, 1154
- Data replication, DDBs, 849–853
- Data security
  - access acceptability and, 1127
  - authenticity assurance and, 1127
  - data availability and, 1127
  - sensitivity of data and, 1126–1127
- Data sources
  - databases as, 425
  - JDBC, 331
- Data striping, RAID, 584–585
- Data transfer costs, DDB query processing, 860–862
- Data types
  - attributes in SQL, 182–184
  - bit strings, 183
  - Boolean, 183
  - character strings, 182–183
  - CREATE TYPE command, 184
  - DATE, 183
  - INTERVAL, 184
  - numeric, 182
  - records, 560–561
  - relational model domains, 151
  - spatial, 989–990
  - TIME, 183
  - TIMESTAMP, 183–184
- Data values, records, 560
- Data warehouses
  - building, 1111–1114
  - data modeling for, 1105–1110
  - defined, 1102
  - ETL (extract, transform, load) process, 1103
  - functionality of, 1114–1115
  - use of, 4
  - views compared to, 1115
- Data warehousing
  - analytical data store (ADS), 1105
  - characteristics of, 1103–1104
  - data mining compared to, 1070
  - DSS (decision-support systems), 1102
  - master data management (MDM), 1110
  - OLAP (online analytical processing), 1102
  - OLTP (online transaction processing), 1102–1103
  - operational data store (ODS), 1105
  - query optimization, 731–733
  - use of, 1101
  - warehouse implementation difficulties, 1115–1117
- Database administrator, *see* DBA (database administrator)
- Database design
  - active databases, 967–972
  - conceptual design, 61–62, 70–72
  - data modal mapping, 62
  - entities and attributes for, 70–72
  - ER (Entity-Relationship) models for, 60–62, 70–72
  - functional requirements for, 61
  - logical design, 62
  - physical design, 62
  - requirements collection and analysis, 60–61
  - schema creation, 61–62
- Database designer, 15
- Database items, transaction processing, 748
- Database management systems, *see* DBMS (database management system)
- Database monitoring, SQL triggers for, 226–227
- Database programming
  - application programming interface (API), 312
  - database application implementation, 309
  - embedding commands in programming language, 311, 314–320
  - evolution of, 309–310
  - impedence mismatch, 312–313
  - language design for, 312, 339
  - library of functions or classes for, 311–312, 326–335
  - overview of techniques and issues, 310–311
  - sequence of interaction, 313–314
  - stored procedures, 335–338
  - Web programming using PHP, 343–359
- Database recovery techniques
  - ARIES recovery algorithm, 827–831
  - caching (buffering) disk blocks, 815–816
  - cascading rollback and, 819–821
  - checkpoints, 818–819, 828–829
  - database backup and recovery from catastrophic failures, 832–833
  - deferred updates for recovery, 814, 821–823
  - force/no-force rules, 817–818
  - fuzzy checkpointing, 819, 828
  - idempotent operations, 815
  - immediate updates for recovery, 815, 823–826
  - multidatabase system recovery, 831–834
  - NO-UNDO/REDO algorithm, 815, 821–823
  - shadow paging, 826–827
  - steal/no-steal rules, 817–818
  - system log for, 814, 817, 818–819
  - transaction rollback and, 819
  - transactions not affecting database, 821
  - UNDO/REDO algorithm, 815, 818
  - write-ahead logging (WAL), 816–818
- Database schema, ontology as, 134

- Database security
  - access acceptability and, 1127
  - access control, 1126
  - additional forms of protection, 1123
  - authenticity assurance and, 1127
  - challenges for maintaining, 1154–1155
  - control measures, 1123–1125
  - data availability and, 1127
  - database administrator (DBA) and, 1125–1126
  - discretionary action control, 1121, 1129–1134
  - discretionary privileges, types of, 1129–1130
  - discretionary security mechanisms, 1123
  - encryption, 1149–1153
  - flow control, 1147–1149
  - GRANT command for, 1131
  - GRANT OPTION for, 1131
  - granting and revoking privileges, 1129–1134
  - information privacy relationship to, 1128–1129
  - label-based security policy, 1139–1140, 1155–1158
  - limiting privilege propagation, 1133–1134
  - mandatory access control (MAC), 1121, 1134–1137
  - mandatory security mechanisms, 1123
  - Oracle, 1155–1158
  - precision compared to security, 1128
  - privacy issues and preservation, 1153–1154
  - privilege specification using views, 1130–1131
  - propagation of privileges, 1131, 1133–1134
  - revoking of privileges, 1131
  - role-based access control (RBAC), 1121, 1137–1139
  - row-level access control, 1139–1140
  - sensitivity of data and, 1126–1127
  - SQL injection, 1143–1146
  - statistical database security, 1146–1147
  - system log modifications and, 1125
  - threats to databases, 1122
  - types of security for, 1122
  - XML access control, 1140–1141
- Database security and authorization subsystem, DBMS, 1123
- Database server, 44
- Database storage
  - organization of, 545–546
  - reorganization, 45
- Database system
  - architectures, 46–51
  - catalog, 10–11, 35, 42–45
  - communication software, 46
  - current state, 35
  - data models, 32–34
  - DBMS classification, 51–53
  - defined, 6
  - environment
    - environment of, 6–7, 42–46
    - extension of, 35
    - initial state, 35
    - instances, 35
    - interfaces, 40–42
    - languages, 38–40
    - module functions in, 31, 42–45
    - populating (loading), 35
    - schemas, 34–38
    - tools, 45–46
    - utilities, 45
    - valid state, 35
- Databases
  - big data storage systems and, 26
  - DBMS (database management systems) for, 6, 9, 17–23, 27
  - active systems, 4, 22
  - application programs for, 6
  - backing up, magnetic tape storage for, 555–556
  - backup and recovery subsystem, 20
  - big data storage, 3
  - characteristics of, 10–14
  - cloud storage, 3
  - constructing, 6, 9
  - data abstraction, 12–13
  - data relationship complexity and, 21
  - database users and, 3–29
  - deductive systems, 22
  - defined, 4
  - development time reduction, 22–23
  - economies of scale, 23
  - employment concerning, 15–17
  - eXtended Markup Language (XML) and, 25
  - extending capabilities of, 25
  - extracting XML documents from, 442–443, 447–453
  - file processing, 10–11
  - flexibility of, 23
  - hierarchical and network systems used as, 23–24
  - history of applications, 23–26
  - information retrieval (IR) systems compared to, 1025–1026
  - integrity constraints, 21–22
  - interchanging Web data, 25
  - maintenance, 6
  - manipulating, 6, 9
  - meta-data, 6, 10
  - multiple user interfaces, 20–21
  - multiple views of, 13
  - multiuser transaction processing, 13–14
  - NOSQL system, 3, 26
  - object-oriented (OODB), 24–25
  - object-oriented systems and, 19
  - online transaction processing (OLTP), 14
  - persistent storage, 19–20
  - program-data independence, 12
  - program-operation independence, 12
  - properties of, 5
  - protection, 6
  - queries, 6, 20
  - real-time technology, 4
  - redundancy control, 18–19
  - relational, 24
  - rules for inferencing information, 22
  - search techniques, 4
  - self-describing data, 10
  - sharing, 6
  - Structured Query Language (SQL), 26
  - standards enforced by, 22
  - traditional applications, 3
  - transactions, 6, 14
  - triggers for, 22
  - unauthorized access restriction, 19
  - updating information, 23
- Datalog language
  - clausal form, 1003–1005
  - deductive databases, 1001, 1002–1003
  - Horn clauses, 1004
  - notation, 1000–1003
  - program safety, 1007–1010
  - queries in, 1004, 1010–1012
- DATE data type, 183
- DBA (database administrators)
  - interfaces for, 42
  - role of, 15
- DBMIN method, transaction processing, 757
- DBMS (database management systems)
  - advantages of approach, 17–23
  - access path options, 52
  - backup and recovery subsystem, 20
  - bottom-tier database server as, 344
  - centralized, 51
  - centralized architecture of, 46–47
  - classification of, 51–53
  - client/server architectures, 47–49
  - component modules, 42–45
  - conceptual design phase, 9
  - concurrency control software for, 13–14
  - data complexity and, 21
  - data models and, 51–53
  - defined, 6
  - disadvantages of, 27
  - distributed, 51
  - federated, 52
  - general purpose, 52
  - heterogeneous, 52
  - homogeneous, 52
  - integrity constraints, 21–22
  - interfaces, 40–42
  - language, 38–40
  - logical design phase, 9
  - multiple user interfaces, 20–21
  - multiuser systems, 51
  - number of sites for, 51–52

- DBMS (*continued*)  
 operators and maintenance personnel, 17  
 persistent storage, 19–20  
 physical design phase, 9  
 query processing, 20  
 redundancy control, 18–19  
 requirements specification and analysis phase, 9  
 single-user systems, 51  
 special purpose, 52  
 SQL and, 177–178  
 stored procedures and, 336–337  
 system designers and implementers, 17  
 tool developers, 17  
 two-tier client-server architecture, 49  
 unauthorized access restriction, 19  
 XML document storage, 442
- DBMS-specific buffer replacement policies, 756–757
- DDBMSs (distributed database management systems)  
 degree of local autonomy, 865–866  
 degree of homogeneity, 865–866  
 technology and, 841  
 update decomposition and, 863–865
- DDBs (distributed databases)  
 advantages of, 846  
 architectures, 868–875  
 autonomy, 845–846  
 availability, 844–845  
 catalog management, 875  
 concurrent control and recovery in, 854–857  
 conditions for, 842–843  
 data allocation, 849–853  
 data fragmentation, 847–853  
 data replication, 849–853  
 network topologies, 843  
 partition tolerance, 845  
 query processing and optimization, 859–865  
 reliability, 844–845  
 scalability, 845  
 sharding, 847–848  
 technology and, 841  
 transaction management in, 857–859  
 transparency, 843–844
- DDL (data definition language)  
 compiler for schema definitions, 42–43  
 DBMS languages and, 39
- Deadlock  
 cautious waiting algorithm, 791  
 detection, 791–792  
 no waiting algorithm, 791  
 occurrence in transactions, 789–790  
 prevention protocols, 790–791  
 timeouts for, 792  
 transaction timestamps and, 790–791
- Debt–credit transactions, 773
- Decision-support systems, *see* DSS (decision-support systems)
- Decision trees, data mining, 1085–1086
- Declaration, XML documents, 433
- Declarative assertions, 225–227
- Declarative expressions, 268
- Declarative languages, 40, 999
- Decomposition  
 algorithms, 519–523  
 Boyce-Codd normal form (BCNF), 489–491, 522–523  
 dependency preservation, 514–515, 519–522  
 DDMS (distributed database management service), 863–865  
 fourth normal form (4NF), 527–530  
 nonadditive (lossless) join property, 476, 515–518, 519–523, 530  
 nonadditive join test for binary decomposition (NJB), 490  
 normalization and, 489–491  
 properties of, 504, 513–518  
 queries, 863–865  
 relations not in BCNF, 489–491  
 three normal form (3NF), 519–522  
 update, 863–865
- Deductive database systems, 22
- Deductive databases  
 clausal form, 1003–1005  
 Datalog language for, 1001, 1002–1003  
 Datalog program safety, 1007–1010  
 Datalog rule, 1004  
 declarative language of, 999  
 enhanced data models, 962, 999–1012  
 Horn clauses, 1004  
 nonrecursive query evaluation, 1010–1012  
 overview of, 999–1000  
 Prolog language for, 1000–1001  
 Prolog/Datalog notation, 1000–1003  
 relational operators for, 1010  
 rules, 1000, 1005–1007
- Deep Web, 1052
- Default values, SQL attributes, 184–186
- Deferred updates, database recovery, 814, 821–823
- Degree of homogeneity, 865–866
- Degree of local autonomy, 865–866
- Degree of relation  
 schema attributes, 152  
 SELECT operations, 243  
 PROJECT operation, 244
- DELETE command, SQL, 200
- Delete operation, relational data models, 166, 167–168
- Deletion, B-Trees, 629–630
- Deletion anomalies, RDB design and, 467
- Deletion marker, files, 568
- Deletion operation, concurrency control and, 806
- Denormalization, 476
- Dependency  
 diagrammatic notation for, 474  
 equivalence of sets of, 508  
 functional, 471–474, 505–512, 527–528, 532  
 inclusion, 531–532  
 inference rules for, 505–509, 527–528  
 join (JD), 494–495, 530–531  
 minimal sets of, 510–512  
 multivalued (MVD), 491–494, 527–530  
 preservation property, 476
- Dependency preservation  
 algorithms, 519–522  
 nonadditive (lossless) join decomposition and, 519–522  
 property of decomposition, 514–515  
 3NF schema using, 519–522
- Dereferencing (→), SQL, 386
- Derived attribute, 66
- Descendant nodes, tree structures, 617
- Description record, SQL/CLI, 327–328
- Descriptors, SQL schemas, 179
- Design, *see* Database design
- Design autonomy, DDBs, 845
- Design transparency, DDBs, 844
- Destructor, object operation, 371
- Dictionary, ontology as, 134
- Dictionary constructor, 369
- Digital certificates, 1153
- Digital libraries, 1047–1048
- Digital signatures, 1152–1153
- Digital terrain analysis, 988–989
- Directed acyclic graph (DAG), 655
- Directed graph, XML data representation, 427–428
- Dirty bit, buffer (cache) management, 558, 816
- Dirty page tables, database recovery, 828–831
- Dirty read problem, transaction processing, 750
- DISCONNECT command, SQL, 316
- Discretionary action control, 1121, 1129–1134
- Discretionary privileges, types of, 1129–1130
- Discretionary security mechanisms, 1123
- Discriminating attributes, 299–300
- Discriminator key, UML class diagrams, 88
- Disjointness constraint (d notation), 114–115
- Disjunctive selection, search methods for, 666–667
- Disk blocks (pages)  
 allocating files on, 564  
 block size, 549–550  
 buffering, 556–560, 815–816  
 database recovery, 815–816  
 hardware addresses of, 550–551  
 interblock gaps in, 550  
 reading/writing data from, 551

- Disk drive, 550, 551–552
- Disk pack, 547
- Disk storage devices
  - capacity of, 547
  - double-sided, 547
  - efficient data access from, 552–553
  - fixed-head, 551
  - formatting, 549–550
  - external hashing, 575–577
  - hardware disk drive (HDD), 547
  - hardware of, 547-
  - interfacing drives with computer systems, 551–552
  - moveable head, 551
  - parameters, 1167–1169
  - RAID, parallelizing access using, 542, 584–588
  - single-sided, 547
- DISTINCT option, SQL, 188, 194
- Distributed computing systems, 841
- Distributed database management systems, *see* DDBMs (distributed database management systems)
- Distributed databases, *see* DDBs (distributed databases)
- Distributed DBMS, 51
- Distributed query processing
  - mapping, 859
  - localization, 859
  - data transfer costs, 860–862
  - semi-join operator, 862–863
- DIVISION operation, 255–257
- Document-based data models, 51, 53
- Document-based NOSQL, 888, 890–895
- Document body specification, HTML, 429
- Document-centric documents, XML, 431
- Document header specifications, HTML, 428
- Document type definition (DTD), XML, 434–436
- Documents
  - data-centric, 431
  - DBMS storage of, 442
  - declaration, XML, 433
  - document-centric, 431
  - extracting from databases, 442–443, 447–453
  - graph-based data for, 447–452
  - hierarchical views of, 447–452
  - hybrid, 431
  - hypertext, 425
  - parentheses for element specifications, 434
  - relational data models for, 447–449
  - schemaless, 432–433
  - schemas, 448–452
  - self-describing, 425
  - storage of, 442–443
  - tags for XML unstructured data, 428–430
  - tree-structured data models for, 431–433, 449–453
  - type of element, 434
  - valid, 434
  - well-formed, 433–424
  - XML, 431–436, 442–443, 447–453
- Domain-key normal form (DKNF), 532–533
- Domain relational calculus
  - formulas (conditions), 277–278
  - join condition, 278
  - nonprocedural language of, 268
  - quantifiers for, 279
  - selection condition, 278
  - variables, 277
- Domain separation (DS) method, transaction processing, 756–757
- Domains
  - atomic values of, 151
  - attribute roles, 152
  - attribute value sets, 69–70
  - cardinality of, 153
  - Cartesian product of, 153
  - constraints, 158
  - data type specification, 151, 184
  - ER model entity types, 69–70
  - format of, 151
  - mathematical relation, 153
  - relation schema and, 152
  - relational data models, 151–152, 158
  - SQL, 184
  - tuples for, 151–152
- Dot notation
  - object operation application, 372, 392
  - path expressions, SQL, 386
  - UDT components, 383
- Double buffering technique, 556–557
- Double-sided disks, 547
- Downgrading locks, 786
- Driver manager, JDBC, 331
- Drivers, JDBC, 331–332
- DROP command, SQL, 233
- DROP TABLE command, SQL, 200
- DROP VIEW command, SQL, 229
- DSS (decision-support systems), 1102
- Duplicates
  - indexes for management of, 641
  - parallel algorithm projection and, 685
  - PROJECT operation elimination of, 245
  - unary operation elimination of, 244–245
- Durability (permanency) property, transactions, 758
- Dynamic arrays, PHP, 345–346
- Dynamic file expansion, hashing for, 577–582
- Dynamic files, 566
- Dynamic hashing, 580
- Dynamic multilevel indexes
  - B-trees and, 601–602, 622–630
  - B-trees and, 601–602, 617–622
  - concept of, 616
  - search trees and, 618–619
  - search, insert and deletion with, 625–629
- Dynamic programming, query optimization and, 716, 725–726
- Dynamic random-access memory (DRAM), 543
- Dynamic spatial operators, 990–991
- Dynamic SQL
  - command preparation and execution, 320–321
  - defined, 310
  - queries specified at runtime, 320–321
- DynamoDB model, 896–867
- e-commerce environment, access control and, 1141
- e-mail servers, client/server architecture, 47
- Early (static) binding, 344
- EER (Enhanced Entity-Relationship) model
  - abstraction concepts, 129–133
  - categories, 108, 120–122, 126
  - class relationships, 108–110
  - conceptual schema refinement, 119–120
  - constraints, 113–116
  - database schema, 122–124
  - design choices, 124–126
  - generalization, 108, 112–120, 124–128
  - hierarchies, 116–119
  - inheritance, 110, 117–119
  - knowledge representation (KR), 128–129
  - lattices, 116–119
  - mapping to ODB schema, 407–408
  - ontology, 129, 132–134
  - semantic data models, 107–108, 129–134
  - specialization, 108, 110–120, 124–128
  - subclasses, 108–110, 117–119, 126
  - superclasses, 109, 110, 117–118, 126
  - UML class diagrams, 127–128
  - union type modeling, 108, 120–122
- EER-to-Relational mapping
  - attributes of relations, 298–300
  - categories, 302–303
  - generalization options, 298–301
  - model constructs to relations, 298–303
  - multiple inheritance and, 301
  - multiple-relation options, 299–300
  - shared subclasses, 301
  - single-relation options, 299–300
  - specialization options, 298–301
  - union types, 302–303
- Element operator, OQL, 413

- Elements
  - complex, XML structure specification, 441
  - empty elements, 440
  - parentheses for specifications of, 434
  - root elements, 440
  - tree-structured data models, 430–431
  - type of in documents, 434, 440–441
  - XML, 430–431, 434, 440–441
- Embedded SQL
  - communication variables in, 315, 316
  - connecting to a database, 315–316
  - cursors for, 317–320
  - database programming approach, 311, 338–339
  - defined, 310, 311
  - host language for, 314
  - Java commands using SQLJ, 321–325
  - precompiler or preprocessor for, 311, 314
  - program variables in, 314–315
  - query results and, 317–320
  - shared variables in, 314
  - tuple retrieval, 311, 314–317
- Empty elements, XML, 440
- Encapsulation
  - constructor function for, 384
  - mutator function for, 384
  - ODBs, 366, 370–374, 384–385
  - object behavior and, 366, 371
  - observer function for, 384
  - operations, 366, 370–374, 384–385
  - object naming and reachability, 373–374
  - SQL, 379–380, 384–385
  - user-defined type (UDT) for, 384–385
- Encryption
  - Advanced Encryption Standards (AES), 1150
  - asymmetric key encryption
    - algorithms, 1151
  - Data Encryption Standards (DES), 1150
  - database security, 1149–1153
  - defined, 1149–1150
  - digital certificates, 1153
  - digital signatures, 1152–1153
  - public key encryption, 1151–1152
  - RSA public key encryption algorithm, 1152
  - symmetric key algorithms, 1150–1151
- End/start tag (</...>), HTML, 428
- End users, 15–16
- Enhanced data models
  - active databases, 963–974
  - active rules, 962, 963–964, 969–973
  - deductive databases, 962, 999–1012
  - functionality and, 961
  - logic databases, 962
  - multimedia databases, 962, 994–999
  - spatial databases, 962, 987–994
  - temporal databases, 962, 974–987
  - temporal querying constructs, 984–986
  - time series data, 986–987
- Enhanced Entity-Relationship model, *see* EER (Enhanced Entity-Relationship) model
- Enterprise flash drives (EFDs), 553
- Entities
  - attributes, 63–70
  - conceptual data modeling, 33
  - conceptual design and, 70–72
  - defined, 63
  - ER mapping of, 291–293
  - ER models and, 63–72, 75, 79
  - generalized, 126
  - identifying (owner) type, 79
  - key (uniqueness constraint) attributes, 68–69, 79
  - NULL values, 66
  - overlapping, 115
  - participation in relationships, 72–73
  - recursive (self-referencing)
    - relationships and, 75
  - role names, 75
  - sets (collection), 67–68
  - strong, 79
  - subclass as, 110, 114–115
  - superclass as, 110
  - types, 67–68, 79, 110
  - value sets (domains) of attributes, 69–70
  - weak, 79, 292–293
- Entity integrity, relational data modeling, 163–165
- Entity-Relationship model, *see* ER (Entity-Relationship) model
- Entrypoints, object names as, 373, 387
- Environment record, SQL/CLI, 327–328
- Environments
  - application programs, 6–7, 46
  - communication software, 46
  - database system, 6–7, 42–46
  - modules, 31, 42–45
  - tools, 45–46
- EQUIJOIN (=) comparison operator, 253
- Equivalence of sets of functional dependency, 508
- Equi-width/equi-height histograms, 713
- ER (Entity-Relationship) diagrams
  - conceptual design choices, 82–84
  - database application use of, 63–64
  - database schema as, 81
  - entity type distinction, 79
  - notations for, 81, 83–88, 1163–1165
  - schema construct names, 82
- ER (Entity-Relationship) model
  - applications of, 59, 62–64, 70–72, 92–94
  - attributes, 63–70
  - constraints on, 73–74, 76–78, 91–92
  - data model type, 33
  - data modeling using, 59–105
  - database design using, 60–62, 80
  - entities, 63–72, 79
  - relationships, 72–78, 88–92
  - schema and, 61–62, 81–85
- Unified Modeling Language (UML)
  - and, 60, 85–88
- Error checking, PHP, 355
- Errors, DDBs, 844
- ER-to-Relational mapping
  - algorithm, 290–296
  - binary relationship types, 293–295
  - entity types, 291–293
  - ER model constructs, 296–298
  - multivalued attributes, 295–296
  - n*-ary relationship types, 296
  - relational database design, 290–298
  - weak entity types, 292–293
- Escape operator (/) in SQL, 196
- ETL (extract, transform, load) process, 1103
- Evaluation for query execution, 701–702
- Event-condition-action (ECA) model
  - active rules (triggers), 963–964
  - SQL trigger components, 227
- Event information versus duration information, 976
- Events, SQL trigger component, 227
- Eventual consistency, NOSQL, 885–886
- EXCEPT operation, SQL sets, 194–195
- Exceptions
  - error handling, 322–323, 393–394
  - ODMG models, 393–394, 397–398
  - operation signature and, 397–398
  - SQLJ, 322–323
- Execution autonomy, DDBs, 845
- Execution for query optimization, 701–712
- Execution transparency, DDBs, 844
- Existence bitmap, 636
- Existence dependency, 77–78
- Existential quantifiers, 271, 274
- EXISTS function, SQL query retrieval, 212–214
- Exists quantifier, OQL, 415
- Expert (knowledge-based) systems, 962–963
- Explicit set of values, SQL, 214
- Expressions
  - Boolean, 241–242
  - declarative, 268
  - formulas and, 270–271
  - in-line, 245
  - relational algebra, 239
  - safe, 276–277
  - tuple relational calculus, 270–271, 276–277
- EXtended Markup Language, *see* XML (EXtended Markup Language)
- Extendible hashing, 578–580
- EXTENDS inheritance, 393
- Extensible Stylesheet Language (XLS), 447
- Extensible Stylesheet Language for Transformations (XSLT), 447
- Extensions, SQL, 178
- Extent inheritance, 377, 385
- Extents
  - class declaration of, 398
  - constraints on, 376–377



- defined, 376
- object persistence and, 373
- ODMG models, 373, 376–377, 398
- persistent collection for, 373, 376
- transient collection for, 376
- type hierarchy and, 376–377
- External hashing, 575–577
- External (schema) level (views), 37
- External sorting, files, 568
- External sorting algorithms, 660–663
- Extraneous attribute, 510
- F*-score, IR, 1046–1047
- Faceted search, IR, 1058–1059
- Fact constellation, 1109
- Fact tables, 1108
- Factory objects, ODMG models, 398–400
- Facts, relation schema and, 156
- Fan-out, multilevel indexes, 613, 622
- Fault, DDBs, 844–845
- Fault tolerance, Big data technology and, 942, 946
- Federated database (FDBS) schema architecture, 871–872
- Federated database system (FDBS), 866–868
- Federated DBMS, 52
- FETCH command, SQL, 317, 319–320
- FETCH INTO command, 325
- Fibre Channel over Ethernet (FCoE), 590–591
- Fibre Channel over IP (FCIP), 590
- Fields
  - connecting, 582–583
  - data type of, 560
  - Fields, records, 560, 561–563, 568–569
  - fixed-length records, 561
  - key, 568
  - mixed records, 582–583
  - optional, 562
  - ordered records, 568–569
  - ordering, 568
  - record type, 583
  - records, 560, 561–563, 568–569
  - repeating, 562–563
  - variable-length records, 561
- Fifth normal form (5NF)
  - definition of, 494
  - functional dependency in, 532
  - join dependency (JD) in, 494–495, 530–531
  - inclusion dependency in, 531–532
- File load factor, hashing, 582
- File processing, 10–11
- File servers, client/server architecture, 47
- Files
  - allocating blocks on a disk, 564
  - B-trees for organization of, 583
  - binary search for, 570
  - clustered files, 572, 583, 602–603
  - data storage using, 541–542
  - database catalog for, 10–11
  - defined, 7
  - dynamic files, 566
  - fully inverted file, 641
  - grid files, 632–633
  - hashing techniques, 572–582
  - headers, 564
  - heaps, 567–568
  - indexed-sequential, 571
  - indexes, 20
  - indexing structures for, 601–652
  - inverted files, 641
  - linear search for, 564, 567–568
  - main (master) files, 571
  - mixed records, 582–583
  - operations on, 564–567
  - ordered (sorted) records, 568–572
  - overflow (transaction), 571
  - records, 560–564, 567–572, 582–583
  - static files, 566
  - storage of, 10–11, 560–572, 582–583
  - unordered records (heaps), 567–568
- Filtering input, SQL injection and, 1146
- First normal form (1NF)
  - atomic (indivisible) values of, 477–478
  - multivalued attributes, 481
  - nested relations, 479–480
  - techniques for relations, 478–479
  - unnest relation, 479–480
- Fixed-head disks, 551
- Fixed-length records, 561–563
- Flag fields, EER-to-relational mapping with, 300
- Flash memory, 543–544
- Flat files, 150
- Flat relational model, 155
- Flow analysis operations, 988
- Flow control, 1147–1149
- FLWOR expression, XQuery, 445
- FOR clause, XQuery, 445–446
- FOR UPDATE OF clause, SQL, 318
- Force/no-force rules, 817–818
- Foreign keys
  - relational data modeling, 163–165
  - SQL constraints, 186–187
  - XML specification, 441
- Formal languages, *see* Relational algebra; Relational calculus
- Format, relational model domains, 151
- Formatting styles, HTML, 428
- Forms-based interfaces, 41
- Forms specification language, 41
- Formulas (conditions)
  - atoms in, 270–271, 277–278
  - Boolean conditions, 270–271
  - domain relational calculus, 277–278
  - tuple relational calculus, 270–271
- Fourth normal form (4NF)
  - decomposition of relations, 529
  - definition of, 493, 528
  - functional dependency and, 527–528
  - inference rules for, 527–528
  - multivalued dependency (MVD) and, 491–494, 527–528
- nonadditive join decomposition into, 530
- normalizing relations, 493–494
- FP-growth algorithm, 1077–1080
- Fragmentation transparency, DDBs, 843–844
- Free-form search request, 1023
- Frequent-pattern (FP) tree, 1077–1080
- FROM clause, SQL, 188–189, 197, 232
- Full functional dependency, 2NF, 481–482
- Fully inverted file, 641
- Function-based indexing, 637–638
- Function call injection, SQL, 1144–1145
- Functional data models, 75
- Functional dependency (FD)
  - Armstrong's axioms, 506–509
  - closure, 505–506, 508
  - defined, 472, 505
  - equivalence of sets of, 508
  - extraneous attribute, 510
  - full functional dependency, 2NF, 481–482
  - inference rules for, 505–509, 527–528
  - left- and right-hand attributes of, 472
  - legal relation states (extensions), 472
  - minimal sets of, 510–512
  - normal forms, 481–483
  - notation for diagrams, 474
  - RDB design and, 471–474, 505–512
  - semantics of attributes and, 472–473
  - transitive dependency, 3NF, 483
  - universal schema relation for, 471–474
- Functional requirements, 61
- Functions
  - aggregate, 216–219, 260–261
  - built-in, 384
  - hashing (randomizing), 572, 580
  - inheritance specifications and, 385
  - overloading, 385
  - PHP programming, 350–352
  - query retrieval and, 216–219
  - relational algebra for, 260–261
  - SQL, 216–219, 384–385
  - type (class) hierarchies and, 374–375
  - UDT, 384–385
  - XML data creation using, 453–455
- Fuzzy checkpointing, 819, 828
- Garbage collection, 827
- Generalization
  - conceptual schema refinement, 119–120
  - constraints on, 113–116
  - defined, 113
  - design choices for, 124–128
  - EER diagram notation for, 112
  - EER modeling concept, 108, 112–120, 124–128
  - entity type, 126
  - hierarchies, 119
  - lattices, 116–119
  - semantic modeling process, 131
  - superclass from subclasses, 112–113
  - total, 115
  - UML notation for, 127–128

- Generalized projection operation, 259–260
- Genetic algorithms (GAs), 1093
- Geographic information systems (GISs), 4, 987
- Global depth, hashing, 578
- Global query optimization, 860
- Global query optimizer, Oracle, 734–735
- Glossary, ontology as, 134
- GRANT command, 1131
- GRANT OPTION, 1131
- Granting and revoking privileges, 1129–1134
- Graph-based data, XML document extraction using, 447–452
- Graph-based data models, 51, 53
- Graph-based NOSQL, 888, 903–909
- Graphical User Interfaces, *see* GUI (Graphical User Interface)
- Grid files, 632–633
- GROUP BY clause
  - SQL, 219–220
  - view merging, subqueries, 705–706
- Grouping
  - aggregate functions and, 216–218, 260–261
  - attributes, 219, 260–261
  - GROUP BY clause for, 219–220
  - HAVING clause for, 219–221
  - NULL values in grouping attributes, 219
  - operator, 415–416
  - OQL, 415–416
  - partitions, 219, 415–416
  - QBE (Query-by-Example) language, 1175–1177
  - relations partitioning into tuples, 219
  - separate groups for tuples, 219
  - SQL query retrieval and, 216–222
  - WHERE clause for, 221–222
- GUI (Graphical User Interface)
  - data mining, 1095
  - DBMS provision of, 20–21
  - use of, 41
- Hadoop
  - advantages of technology, 936
  - Big data technology for, 916–917, 921–926
  - distributed file system (HDFS), 921–926
  - ecosystem, 926
  - historical background of, 916–917
  - parallel RDBMS compared to, 944–946
  - releases, 921
  - YARN (Hadoop v2), 936–944, 949–953
- Handles, SQL/CLI records, 328
- Handle variables, SQL/CLI declaration of, 328
- Hardware
  - addresses, 550–551
  - disk storage devices, 547–552
- Hash field, 572
- Hash file, 572
- Hash (randomizing) functions, 572, 580
- Hash indexes, 633–634
- Hash key, 572
- Hash partitioning, 684
- Hash tables, 572–573
- Hashing techniques
  - dynamic file expansion, 577–582
  - dynamic hashing, 580
  - extendible hashing, 578–580
  - external hashing, 575–577
  - file storage, 572–582
  - folding, 574
  - internal hashing, 572–575
  - linear hashing, 580–582
  - multiple keys and, 632
  - partitioned hashing, 632
  - static hashing, 577
- Having clause, OQL, 416
- HAVING clause, SQL, 219–221
- Hbase data model
  - column based systems, 900–903
  - CRUD operations, 903
  - distributed system concepts for, 903
  - NOSQL, 900–903
  - versioning, 900–902
- Headers, file descriptors, 564
- Heaps (unordered file records), 567–568
- Here documents, PHP, 347–348
- Heterogeneous DBMS, 52
- Heuristic rules for query optimization, 657, 692, 697–701
- Hidden attributes, objects, 371, 375
- Hierarchical data models, 33, 53. *See also* Tree-structured data models
- Hierarchical systems using databases, 23–24
- Hierarchical views, XML document extraction using, 447–453
- Hierarchies
  - association rules for data mining, 1081–1082
  - EER models, 116–119
  - generalization, 119
  - inheritance and, 118
  - memory, 543–545
  - object data models (acyclic graphs), 52
  - specialization, 116–119
  - tree structure, 116, 452–453
  - type (class), 366, 374–377, 385
- High-level (conceptual) data models, 33, 60–62
- High-level (nonprocedural) DML, 39–40
- High-level language support, Big data technology and, 946
- High-performance data access, NOSQL, 886–887
- Hints, Oracle, 736
- Histograms
  - cost estimation from, 713
  - equi-width/equi-height, 713
  - selection conditions and, 668
- HITS ranking algorithm, 1051
- HOLAP (hybrid OLAP) option, 1114
- Homogeneous DBMS, 52
- Horizontal fragmentation (sharding), DDB data, 843–844, 847–848
- Horizontal partitioning, 684
- Horn clauses, 1004
- Host language, embedded SQL, 314
- Hot set method, transaction processing, 757
- Hoya (Hortonworks HBase on YARN), 943–944
- HTML (HyperText Markup Language)
  - client tier of, 344
  - tag notation and use, 428–430
  - Web data and, 25
- HTML tag (<...>), 428
- Hybrid documents, XML, 431
- Hybrid-hash join, 675–676
- Hyperlinks, 25, 1027
- Hypertext documents, 425
- HyperText Markup Language, *see* HTML (HyperText Markup Language)
- Idempotent operations, 815
- Identification, semantic modeling process, 130–131
- Identifying (owner) entity type and relationship, 79
- Image data, 989
- Images
  - automatic analysis, 996–997
  - color, 997
  - defined, 995
  - multimedia databases for, 995–999
  - object recognition, 997–998
  - semantic tagging of, 998–999
  - shape, 997
  - texture, 997
- Immediate updates
  - database recovery, 815, 823–826
  - SQL views, 230
- Immutable property of OID, 367
- Impedence mismatch, 312–313
- Implementation
  - active databases, 967–972
  - aggregate operations, 678–679
  - database operations, 12
  - JOIN operations for, 668–681
  - operation encapsulation and, 371
  - pipelining using iterators, 682–683
  - query processing, 668–676, 679–681
  - temporal databases, 982
- Implementation (physical storage) level, RDB design, 459–460
- IN comparison operator, SQL, 209–210
- In-line expression, 245
- In-line views, SQL, 232
- In-place updating, 816
- Inclusion dependency, 5NF, 531–532
- Incorporating time, temporal databases, 977–984
- Incorrect summary problem, transaction processing, 750
- Incremental updates, SQL views, 230
- Incremental view maintenance, 707–710



- Index-based nested-loop join, 559, 718–719
- Indexed allocation, file blocks, 564
- Indexed (ordered) collection expressions, OQL, 415
- Indexed-sequential file, 571, 616
- Indexes
  - bitmap indexes, 634–637
  - clustering, 602, 606–608
  - constraint management using, 641
  - creation of, 639–640
  - data modeling access path, 34
  - DBMS auxiliary files, 20
  - duplicate management using, 641
  - fully inverted file, 641
  - hash indexes, 633–634
  - locks for concurrency control, 805–806
  - logical versus physical, 638–639
  - multilevel, 613–617
  - multiple keys for, 613–633
  - ordered index on multiple attributes, 631–632
  - physical database file structures as, 641
  - primary, 602, 603–606
  - rebuilding, 640
  - secondary, 603, 609–612
  - single-level ordered, 602–613
  - spatial data, 991–993
  - SQL creation of, 201–202
  - tuning, 640–641
- Indexing fields, 601, 602
- Indexing structures
  - column-based storage of relations, 642
  - hints in queries, 641–642
  - physical database design and, 601–652
  - ordered sequential access method (ISAM), 601
  - B-trees, 601–602, 622–630, 636–637
  - B-trees, 601–602, 617–622, 629–630
  - single-level ordered indexes, 602–613
  - multilevel indexes, 613–617
  - multiple keys for, 631–633
  - hash indexes, 633–634
  - bitmap indexes, 634–637
  - function-based indexing, 637–638
  - issues concerning, 638–642
  - RDB design and, 643–646
  - strings, 640
- Industrial internet of things (IIOT or IOT), 914
- Inference engine, deductive databases, 999, 1004–1005
- Inference rules
  - Armstrong's axioms, 506–509
  - closure, 505–506, 508
  - 4NF schema using, 527–528
  - functional dependencies, 505–509, 527–528
  - proof by contradiction, 507
  - multivalued dependencies, 527–528
- Information extraction (IE), 1040
- Information privacy, security
  - relationship to, 1128–1129
- Information repository, DBMS, 46
- Information retrieval (IR)
  - Boolean model, 1030
  - data, 1024
  - databases compared to IR systems, 1025–1026
  - defined, 1022–1023
  - desktop search engines for, 1025
  - enterprise search systems for, 1024
  - F*-score for, 1046–1047
  - free-form search request, 1023
  - history of, 1026–1027
  - information need, 1024
  - inverted indexing, 1040–1044
  - levels of scale, 1024
  - modes of interaction in IR systems, 1027–1028
  - pipeline for processing, 1028–1029
  - probabilistic model, 1033–1034
  - queries in IR systems, 1035–1037
  - recall and precision, 1044–1046
  - search relevance, 1044–1047
  - semantic approach, 1028
  - semantic model, 1034–1035
  - statistical approach, 1028
  - text preprocessing, 1037–1040
  - trends in, 1057–1063
  - unstructured information, 1022
  - users, 1023–1024
  - vector space model, 1031–1033
- Information updating, 23
- Inherent model-based (implicit) constraints, 157
- Inherent rules, 22
- Inheritance
  - behavior inheritance, 393
  - class–schema interface, ODL, 404–405
  - colon (:) notation for, 393
  - EER-to-relational mapping, 301
  - EXTENDS, 393
  - extent inheritance, 377, 385
  - function overloading and, 385
  - generalization lattice or hierarchy, 119
  - interface inheritance, 377, 393
  - multiple, 118, 301, 377–378, 393
  - ODBs, 366, 374–377, 377–378, 385, 393
  - ODMG object model and, 393, 404–405
  - selective, 377
  - simplified model for, 347–377
  - single, 118–119
  - specialization lattice or hierarchy, 117–118
  - SQL, 380
  - subclass/superclass relationships, 110, 117–119
  - table inheritance, 385
  - type inheritance, 385
- Initial hash function, 580
- Initial state, populating (loading) databases and, 35
- Inner join, SQL table (relations), 215–216
- Inner/outer joins, 254, 263–264
- Innermost nested query, 211
- INSERT command, SQL, 198–200
- Insert operation
  - constraint violations and, 166–167
  - relational data models, 166–167
- Insertion, B-trees, 626–629
- Insertion anomalies, RDB design and, 465–466
- Instance variables, 365–366
- Instances (occurrences), 35, 72
- Instantiable class behavior, interface and, 392
- Instantiation, semantic modeling process, 130
- Integrity constraints
  - database applications and, 21–22
  - entity integrity, 163–165
  - foreign keys and, 163–164
  - referential integrity, 21, 163–165
  - relational modeling and, 160–165
  - relational database schemas and, 160–163
  - semantic, 165
  - valid and not valid states and, 160–161
- Intellectual property rights, 1154–1155
- Intention, 35
- Interactive query interface, 43–44
- Interactive transactions, concurrency control and, 807
- Interblock gaps, disk devices, 550
- Interface inheritance, 377, 393
- Interfaces. *See also* GUI (Graphical User Interfaces)
  - built-in, ODMG models, 393–396
  - class–schema inheritance, ODL, 404–405
  - database operations, 12
  - DBMS, 20–21, 40–42
  - disk drives with computer systems, 551–552
  - instantiable class behavior and, 392
  - multiple user, 20–21
  - noninstantiable object behavior and, 392
  - object model definitions, 389–392
  - ODMG models and, 389–396, 404–405
  - operation encapsulation and, 371
  - operation specifications, 366
- Interleaved concurrency, 747
- Interleaved processes, 747
- Internal hashing, 572–575
- Internal (schema) level, 36
- Internal nodes, tree structures, 622
- Internet SCSI (iSCSI), 590
- Interpolating variables within text strings, 347
- Interpreted queries, 710
- Interquery parallelism, 687

- INTERSECT operation, SQL sets, 194–195
- INTERSECTION operation, 247–249
- INTERVAL data type, 184
- INTO clause, 317
- Intraquery parallelism, 687
- inverse references, 366, 370, 396–397
- Inverse relationships, ODMG objects, 396–397
- Inverted files, 641
- Inverted indexing
  - construction of, 1041–1042
  - defined, 1041
  - information retrieval (IR), 1040–1044
  - Lucern indexing/search engine for, 1043–1044
  - process of, 1042
- IS-A relationship, 109, 126
- IS/IS NOT comparison operators, 209
- Isolation. *See also* Snapshot isolation
  - levels of in transactions, 758
  - property, transactions, 14, 158
- Iterator object, ODMG models, 393
- Iterator variables
  - query results and, 312
  - OQL, 409–410
- Iterators
  - defined, 682
  - pipelining implementation using, 682–683
  - SQLJ query result processing with, 323–325
- Java
  - embedding SQL commands (SQLJ), 321–325
  - exceptions for error handling, 322–323
  - Web programming technologies, 358
- Java server pages (JSP), 358
- Java servlets, 358
- JavaScript, 358
- JavaScript Object Notation (JSON), 358
- JDBC (Java Database Connectivity)
  - class library imported from, 331, 332
  - drivers, 331–332
  - programming steps, 332–335
  - SQL class library, 326, 331–335
  - two-tier client/server architecture and, 49
- Join attribute, 253
- Join condition, 189, 191, 252, 278
- Join dependency (JD), 5NF, 494–495
- JOIN operations
  - aggregate operation implementation and, 678–679
  - anti-join (AJ) operator, 658–660, 677–678, 681, 719–720
  - attributes, 668
  - bucket join, 931
  - buffer space and, 672–673
  - cardinality, 719–720
  - cost functions for, 717–726
  - distributed query processing, 862–863
  - dynamic programming approach to ordering, 725–726
  - EQUIJOIN (=) comparison operator, 253
  - hybrid-hash join, 675–676
  - index-based nested-loop join, 559, 718–719
  - inner/outer, 254, 263–264
  - join selectivity (js) operator, 717–718
  - MapReduce (MR), 930–932
  - map-side hash join, 930
  - multiway joins, 668
  - N-way joins, 931–932
  - NATURAL JOIN (\*\_\_ ) comparison operator, 253, 262–263
  - nested-loop join, 558–559, 672–673, 718
  - non-equi-join, 681
  - optimization based on cost formulas, 720–721
  - ordering choices in multirelational queries, 721–724
  - OUTER JOIN operations, 262–264, 679–681
  - parallel algorithms, 685–686
  - partition-hash join, 559, 674–675, 719, 930–931
  - performance of, 673–674
  - physical optimization, 724
  - query processing implementation, 668–676, 679–681
  - recursive closure operations, 262
  - relational algebra and, 251–255, 262–264
  - semi-join (SJ) operator, 658–660, 681, 719–720, 862–863
  - SQL query retrieval, 215–216
  - SQL relations, 215–216
  - sort-merge join, 559, 719, 930
  - two-way join, 668
- k-means algorithm, 1088–1089
- Key constraints
  - attributes, 68–69, 302
  - database integrity and, 21
  - integrity constraints and, 163–165
  - referential integrity constraints and, 163–165
  - relational modeling and, 158–160, 163–165
  - relational schema and, 157–165
  - surrogate, 302
  - uniqueness property, 68–69, 159
- Key field, records, 568
- Key-value storage (data models), 34, 51, 53
- Key-value stores, NOSQL, 888, 895–900
- Keys
  - attributes, 477
  - candidate key, 159–160, 477
  - composite keys, 631
  - defined, 476
  - foreign keys, 163–165, 186–187
  - indexes with, 631–633
  - multiple keys, 631–633
  - normal forms and, 476–477
  - ODMG object model, 398
  - primary key, 159, 186–187, 441, 477
  - SQL, 186–187
  - superkey, 158–159, 476–477
  - unique keys, 160
  - XML schema specification, 441
- Keyword-based data search, 41
- Keyword queries, 1035
- Knowledge discovery in databases (KDD), 1070–1073
- Knowledge representation (KR)
  - abstraction concepts, 129–133
  - domain of knowledge for, 129
  - EER modeling and, 128–129
  - ontology and, 129
  - reasoning mechanisms, 129
- Label-based security policy
  - architecture, 1156–1157
  - multilevel security, 1139–1140
  - Oracle, 1155–1158
  - Virtual private database (VPD) technology, 1156
- Language design for database programming, 312, 339
- Latches, concurrency control and, 807
- Late (dynamic) binding, 377
- Lattices
  - EER models, 116–119
  - generalization, 119
  - inheritance and, 117–118
  - specialization, 116–119
- Lazy updates, SQL views, 230
- Leaf class, 127
- Leaf nodes, tree structures, 257, 617, 623
- Least recently used (LRU) strategy, buffering, 559
- Legacy data models, 33, 51, 53
- Legal relation states (extensions), 472
- Level trigger, 967
- Library of functions or classes
  - application programming interface (API), 312, 326
  - database programming approach, 311, 338–339
  - JDBC: SQL class library, 326, 331–335
  - SQL/CLI (SQI call level interface), 326–331
- Lifetime of an object, 388
- LIKE comparison operator, SQL, 195–196
- Linear hashing, 580–582
- Linear regression, data mining, 1092
- Linear scale-up, 684
- Linear search, files, 564, 567–568
- Linear speed-up, 684
- Link structure analysis, Web search and, 1050–1051
- Linked allocation, file blocks, 564

- Links, UML class diagrams, 87
- List constructor, 369
- Literal declaration, 392
- Literals
  - atomic (single-valued) types, 368, 388
  - collection, 392
  - constructors for, 368–370
  - deductive databases, 1002–1003
  - objects compared to, 368
  - ODBs, 368–370, 388, 392
  - ODMG models, 388, 392
  - structured, 388
  - type generators, 368–369
  - type structures for, 368–370
- Loading utility, 45
- Local area network, 842
- Local depth, hashing, 578
- Local query optimization, 860
- Localization, DDB query processing, 859
- Location analysis, 988
- Location transparency, DDBs, 843
- Locking data items, 781
- Locks
  - binary locks, 782–784
  - certify locks, 796–797
  - concurrency control and, 782–786, 796–797, 805–806
  - conversion of, 786
  - downgrading, 786
  - index concurrency control using, 805–806
  - shared/exclusive (read/write) locks, 784–786
  - upgrading, 786, 797
- Log buffers, 755, 756
- Log sequence number (LSN), 828
- Logic databases, 962
- Logical (conceptual) level, RDB design, 459–460
- Logical comparison operators, SQL, 188–190
- Logical data independence, 37–38
- Logical database design, *see* Data model mapping
- Logical design, 62
- Logical index, 638–639
- Logical theory, ontology as, 134
- Loss of confidentiality, database threat of, 1122
- Loss of integrity, database threat of, 1122
- Lossy design, 515
- Lost update problem, transaction processing, 750
- Low-level (physical) data models, 33–34
- Low-level (procedural) DML, 40
- Lucern indexing/search engine, 1043–1044
- Magnetic tape
  - backing up databases using, 555–556
  - memory hierarchy and, 544–545
  - storage devices, 555–556
  - tape reel, 555
- Main (master) file, 571
- Main memory, 543
- Maintenance, databases, 6
- Maintenance personnel, 17
- Mandatory access control (MAC), 1121, 1134–1137
- Mandatory security mechanisms, 1123
- Map data, 989
- Mappings
  - data model, 62
  - database schema views, 37
  - distributed query processing, 859
  - EER model constructs to relations, 298–303
  - EER schema to ODB schema, 407–408
  - ER-to-relational, 290–298
  - ODB conceptual design, 407–408
  - tuples for relations, 154
- MapReduce (MR)
  - advantages of technology, 936
  - Big data technology for, 917–921, 926–936
  - historical background of, 917–918
  - joins in, 930–932
  - parallel RDBMS compared to, 944–946
  - programming model, 918–921
  - runtime, 927–930
- Map-side hash join, MapReduce (MR), 930
- Mark up, XML documents for HTML, 428–429
- Market-basket data model, 1073–1075
- Mass storage, 543
- Master data management (MDM), 1110
- Master-master replication, NOSQL, 886
- Master-slave replication, NOSQL, 886
- Materialized evaluation, 681, 702–702
- Materialized views, query execution, 707–710
- Mathematical relation, domains, 152
- MAX function, SQL, 217
- MAXIMUM function, grouping, 260
- Measurement operations, 988
- Mechanical arm, disk devices, 551
- Memory
  - cache, 543
  - dynamic random-access (DRAM), 543
  - flash memory, 543–544
  - hierarchies, 543–545
  - magnetic tape, 544–545
  - main, 543
  - optical drives, 544
  - random-access (RAM), 543
  - storage capacity and, 543
  - storage devices for, 543–545
- Menu-based interfaces, 40
- Merging phase, external algorithms, 661
- Meta-data
  - database catalog and, 10–11
  - defined, 6
  - schema storage, 35
- Methods
  - database operations, 12
  - object data models, 53
  - operation implementation and, 366, 371
- Middle-tier Web server, PHP as, 344
- Middleware layer, *n*-tier architecture, 50–51
- MIN function, SQL, 217
- Minimal sets of functional dependency, 510–512
- MINIMUM function, grouping, 260
- Miniworld, 5
- MINUS operation, 247–249
- Mirroring, (shadowing), RAID, 585
- Mixed (hybrid) fragmentation, DDB data, 847–848
- Mixed records, files for, 582–583
- Mobile applications, access control of, 1141–1142
- Mobile device apps
  - ER modeling and, 59
  - interfacing, 40–41
  - user transactions by, 16
- Model-theoretic interpretation of rules, 1005
- Models, *see* Data models; EER (Enhanced Entity-Relationship) model; ER (Entity-Relationship) model; Object data models
- Modification anomalies, RDB design and, 467
- Modifier, object operations, 371
- Modules
  - buffering (caching), 20, 42
  - client module, 31
  - compilers, 42–45
  - database queries and, 20, 43–44
  - database systems, 31, 42–45
  - DBMS components, 42–45
  - interactive query interface, 43–44
  - server module, 31
  - stored data manager, 42
- MOLAP (multidimensional OLAP) function, 1114
- MongoDB data model
  - CRUD operations, 893
  - documents, 890–893
  - NOSQL, 890–895
  - replication in, 894
  - sharding in, 894–895
- Moveable head disks, 551
- Multidatabase system recovery, 831–834
- Multidimensional models, 1108
- Multilevel indexes
  - dynamic, 616, 617–630
  - fan-out, 613, 622
  - levels, 613–616
  - physical database design and, 613–617

- Multimedia databases
  - audio data source analysis, 999
  - concepts, 994–996
  - enhanced data models, 962, 994–999
  - image automatic analysis, 996–997
  - object recognition, 997–998
  - semantic tagging of images, 998–999
  - types of, 3–4
- Multiple granularity locking
  - concurrency control and, 801–804
  - granularity levels for, 801
  - granularity of data items, 800–801
  - protocol, 802–804
- Multiple hashing, collision resolution, 575
- Multiple inheritance, 118, 301, 377–378, 393
- Multiple keys
  - grid files and, 632–633
  - indexes on, 613–633
  - multiple attributes and, 631–632
  - ordered index on, 631–632
  - partitioned hashing with, 632
  - physical database design and, 613–633
- Multiple-relation options, EER-to-relation mapping, 299–300
- Multiple user interfaces, 20–21
- Multiplicities, UML class diagrams, 87
- Multiprogramming
  - concept of, 746–747
  - operating systems, 747
- Multirelational queries, JOIN ordering
  - choices and, 721–724
- Multiset (tuple) operations
  - comparisons for query retrieval, 209–211
  - SQL tables, 193–195
- Multiuser DBMS systems, 51
- Multiuser transaction processing, 13–14
- Multivalued attributes, 66, 295–296, 481
- Multivalued dependency, *see* MVD (multivalued dependency)
- Multiversion concurrency control, 781, 795–797
  - certify locks for, 796–797
  - timestamp ordering (TO), 796
  - two-phase locking (2PL), 796–797
- Multiway joins
  - implementing, 668
  - SQL table (relations), 216
- Mutator function, SQL encapsulation, 384
- MVD (multivalued dependency)
  - all-key relation of, 491, 493
  - definition of, 491–492
  - fourth normal form (4NF) and, 491–494, 527–530
  - inference rules for, 527–528
  - normalizing relations, 493–494
  - trivial/nontrivial, 493
- n*-ary relationship types, mapping of, 296
- n*-degree relationships, 88–92
- n*-tier architecture for Web applications, 49–51
- N*-way joins, MapReduce (MR), 931–932
- Named iterator, SQLJ, 323
- Namespace, XML, 440
- Naming mechanisms
  - constraints, SQL, 187
  - database entrypoints, 373
  - object persistence and, 373–374
  - operations for renaming attributes, 245–246
  - query retrieval and, 192, 214–215
  - renaming attributes, 192, 214–215, 245–246
  - schema constructs, 82
- Naming transparency, DDBs, 843
- NATURAL JOIN (\*    ) comparison operator, 253, 262–263
- NATURAL JOIN operation, SQL tables, 215
- Natural language interfaces, 41
- Natural language queries, 1037
- Neo4j system
  - cypher query language of, 905–908
  - distributed system concepts for, 908–909
  - nodes, 904–905
  - NOSQL, 903–909
  - relationships, 904–905
- Nested-loop join, 558–559, 672–673, 718
- Nested queries
  - comparison operators for, 210–211
  - correlated, 211–212
  - innermost query of, 211
  - outer query of, 209
  - query optimization and, 702–704
  - subqueries, 702–704
  - tuple values in, 209–211
  - unnesting (decorrelation), 704
- Nested relations, 1NF in, 479–480
- Network-attached storage (NAS), 589–590
- Network data models, 33, 51, 53
- Network systems using databases, 23–24
- Network topologies, 843
- Neural networks, data mining, 1092
- No waiting algorithm, deadlock prevention, 791
- NodeManager, YARN, 942
- Nodes
  - constant, query graphs, 273
  - leaf, query trees, 257
  - relation, query graphs, 273
  - tree structures, 617
- Non-equi join implementation, 681
- Nonadditive (lossless) join property algorithms, 519–523
  - Boyce-Codd normal form (BCNF) schemas using, 522–523
  - dependency preservation and, 519–522
  - 4NF schema using, 530
  - normalization process, 476
  - RDB decomposition, 515–518, 519–522
  - successive decompositions, 517–518
  - testing binary decompositions for, 517
  - 3NF schema using, 519–522
- Nonadditive join test for binary decomposition (NJB), 490
- Noninstantiable object behavior, interface and, 392
- Nonprocedural language, 268
- Nonrecursive query evaluation, 1010–1012
- Nonserial schedules, 763, 764–765
- Normal form test, 475
- Normal forms
  - Boyce-Codd normal form (BCNF), 487–491
  - defined, 475
  - denormalization, 476
  - domain-key (DKNF), 532–533
  - fifth normal form (5NF), 494–495
  - first normal form (1NF), 477–481
  - fourth normal form (4NF), 491–494
  - insufficiency of for relational decomposition, 513–514
  - join dependency (JD) and, 494–495
  - keys, attributes and definitions for, 476–477
  - multivalued dependency (MVD) and, 491–494
  - normalization of relations, 474–476, 482, 485, 486–487, 493–494
  - practical use of, 476
  - primary keys for, 483–495
  - RDB design and, 474–495, 513–514, 528–533
  - second normal form (2NF), 481–482, 484–486
  - third normal form (3NF), 483–484, 486–487
- Normalization process
  - algorithms, 519–527
  - data normalization, 475–476
  - dependency preservation property, 476
  - multivalued dependency (MVD), 493–494
  - nonadditive (lossless) join property, 476
  - normal form test for, 475
  - relations, 474–476
- NOSQL database system
  - availability, 885–886
  - big data storage uses, 3, 26
  - CAP theorem, 888–890
  - categories of, 887–888
  - column-based, 888, 900–903
  - CRUD (create, read, update, and delete) operations, 887, 893, 903
  - data models, 34, 51
  - DDB similar characteristics, 885–887
  - distributed storage using, 883
  - document-based, 888, 890–895
  - emergence of, 884–885
  - eventual consistency, 885–886
  - graph-based, 888, 903–909
  - Hbase data model, 900–903

- high-performance data access, 886–887
- key-value stores, 888, 895–900
- MongoDB data model for, 890–895
- Neo4j system, 903–909
- query language similar characteristics, 887
- replication models for, 886
- replication, 885–886, 894
- scalability, 885
- sharding, 886, 894–895
- versioning, 887, 899, 900–902
- NOT FINAL, UDT inheritance specification, 385
- NOT operator, *see* AND/OR/NOT operators
- NO-UNDO/REDO algorithm, 815, 821–823
- NULL values
  - aggregate functions and, 218
  - attribute not applicable, 208
  - complex query retrieval and, 208–209
  - constraints on attributes, 160, 184–186
  - discarded values, 218
  - entity attributes, 66
  - grouping attributes with, 219
  - IS/IS NOT comparison operators for, 209
  - query retrieval in SQL, 208–209, 218, 219
  - RDB design problems, 523–524
  - referential integrity and, 163–164
  - relational modeling and, 155–156, 160
  - relation schema for RDB design and, 467–468
  - grouping attributes, 219
  - SQL attribute constraints, 184–186
  - three-valued logic for comparisons, 208–209
  - tuples for relations, 155–156, 163, 467–468
  - unavailable (or withheld) value, 208
  - unknown value, 208
- Numeric arrays, PHP, 349
- Numeric data types, 182, 348
- Object-based storage, 591–592
- Object Data Management Group, *see* ODMG (Object Data Management Group)
- Object data models
  - classes, 52
  - data model type, 33
  - DBMS classification from, 51, 52–53
  - hierarchies (acyclic graphs), 52
  - methods, 53
  - ODMG, 387–400
- Object databases, *see* ODBs (object databases)
- Object definition language, *see* ODL (object definition language)
- Object identifier, *see* OID (object identifier)
- Object identity
  - literal values for, 368
  - ODBs, 367–368, 378
  - OID implementation of, 367
  - SQL, 379
- Object-oriented systems, persistent storage, 19–20
- Object query language, *see* OQL (object query language)
- Object recognition, multimedia databases, 997–998
- Object-relational systems
  - extended-relational systems, 53
  - SQL, 202
- Objects
  - arrow ( $\rightarrow$ ) notation for, 392
  - atomic (single-valued) types, 368, 388, 396–398
  - attributes, 396
  - behavior of based on operations, 371
  - collections, 373, 376
  - constructors for, 368–370
  - dot notation for, 372, 392
  - encapsulation of, 366, 371
  - exceptions, 397–398
  - hidden attributes, 371
  - instance variables, 365–366
  - interfaces, noninstantiable behavior and, 392
  - lifetime, 388
  - literals compared to, 368
  - naming, 373–374, 387
  - ODBs, 365–371, 387–388, 395–400
  - ODMG models, 387–388, 392, 395–400
  - operations for, 370–372
  - persistent, 365, 373–374, 376
  - reachability, 373–374
  - relationships, 396–397
  - signatures, 366, 397
  - state of, 387
  - structure of, 388
  - transient, 365, 373, 376
  - type generators, 368–369
  - type structures for, 368–370
  - unique identity, 367–368
  - visible/hidden attributes, 371, 375
- Observer function, SQL encapsulation, 384
- ODBC (Open Database Connectivity)
  - data mining, 1094–1095
  - standard, 49, 326
- ODBs (object databases)
  - C++ language binding, 417–418
  - conceptual design, 405–408
  - development of, 363–365
  - encapsulation of operations, 366, 370–374, 384–385
  - inheritance and, 366, 374–377, 378, 385, 393
  - instance variables, 365–366
  - inverse references, 366, 370, 396–397
- literals in, 368–370, 388–392
- Object Data Management Group (ODGM) model, 386–405, 417–418
- object definition language (ODL) and, 386, 400–405
- object identifier (OID), 367–368
- object query language (OQL), 408–416
- object-oriented (OO) concepts, 365–366
- objects in, 365–371, 387–388, 395–400
- polymorphism (operator overloading), 366, 377
- RDB compared to, 405–406
- SQL extended from, 379–386
- type (class) hierarchy, 366, 374–377
- ODL (object definition language)
  - classes, 400, 404–405
  - class-schema interface inheritance, 401–404
  - Object Data Management Group (ODGM) model and, 386, 400–405
  - object databases (ODBs) and, 386–387, 400–405
  - schemas, 400–403
  - type constructors in, 369
- ODMG (Object Data Management Group)
  - atomic (user-defined) objects, 395–398
  - bindings, 386, 417–418
  - built-in interfaces and classes, 393–396
  - C++ language binding, 386, 417–418
  - database standard, 33, 364–365
  - extents, 373, 376–377, 398
  - factory objects, 398–400
  - inheritance in object models, 393
  - interface definitions for object models, 389–392
  - keys, 398
  - literals in object models, 388, 392
  - object databases (ODBs), 386–405, 417–418
  - object definition language (ODL) and, 386, 400–405
  - object model of, 387–400
  - object query language (OQL) and, 386, 408
  - objects, 387–388, 392, 395–400
  - standards, 386, 417–417
- OID (object identifier)
  - immutable property of, 367
  - ODB unique object identity and, 367–368
  - ODMG models, 387
  - reference types used for in SQL, 383
- OLAP (Online analytical processing)
  - data warehousing and, 1102
  - data warehousing characteristics and, 1104
  - HOLAP (hybrid OLAP) option, 1114
  - MOLAP (multidimensional OLAP) function, 1114
  - ROLAP (relational OLAP) function, 1114
  - use of, 4



- OLTP (online transaction processing)
  - data warehousing and, 1102
  - multiuser transaction processing, 14
  - relational data modeling, 169
  - special-purpose DBMS use, 52
- Online analytical processing, *see* OLAP (Online analytical processing)
- Online transaction processing, *see* OLTP (online transaction processing)
- Ontology
  - conceptualization and, 134
  - defined, 134
  - knowledge representation (KR) and, 129
  - semantic Web data models, 133–134
  - specification and, 134
  - types of, 134
- Ontology-based information integration, 1052–1053
- OO (object-oriented) concepts, 365–366
- OODB (object-oriented database)
  - attribute versioning, 982–984
  - database complexity and, 24–25
  - development of, 363
  - temporal databases incorporating time in, 982–984
- OQL (object query language)
  - aggregate functions, 413–414
  - Boolean (true/false) results, 414
  - collection operators, 413–416
  - element operator, 413
  - exists quantifier, 415
  - grouping operator, 415–416
  - indexed (ordered) collection expressions, 415
  - iterator variables for, 409–410
  - named query specification, 412–413
  - ODBs, 408–416
  - ODGM model queries and, 408–416
  - ODMG standard and, 386
  - path expressions, 410–412
  - query results, 410–412
  - select...from...where structure of, 409
- OOPL (object-oriented programming language), class library for, 312
- op comparison operator, 270
- Open addressing, hashing collision resolution, 574
- OPEN CURSOR command, SQL, 317
- OpenPGP (Pretty Good Privacy) protocol, XML, 1140–1141
- Operating system (OS), 42
- Operational data store (ODS), 583, 1105
- Operations. *See also* Query processing strategies
  - aggregate, 678–679
  - assignment (←) for, 245
  - binary, 240, 251–259, 262–264
  - defined, 12
  - delete, 166, 167–168
  - dot notation for objects, 372
  - encapsulation, 366, 370–374, 384–385
  - files, 564–567
  - generalized projection, 259–260
  - insert, 166–167
  - JOIN, 251–255, 262–264, 668–676
  - method (body) of, 366, 371
  - ODBs, 366, 370–374, 384–385
  - pipelining for combinations of, 681–683
  - program variables for, 565–566
  - record-at-a-time, 566
  - recursive closure, 262
  - relational algebra, 240–259, 262–265
  - relational data modeling, 165–168
  - renaming attributes, 245–246
  - retrievals, 165–166, 564–565
  - schedules, 759–760, 773
  - selection conditions for, 564–565
  - sequence of, 245–246
  - set-at-a-time, 566
  - set theory and, 246–251, 264–265
  - signature (interface) of, 366, 371
  - SQL query recovery and, 194–197
  - SQL sets, 194–195
  - unary, 240, 241–246
  - UNION, 194–195, 264–265
  - update (modify), 166, 168–169, 564–565
  - user-defined functional requirements, 61
- Operator-level parallelism, 684–686
- Operators
  - aggregate functions, 216–219, 260–261
  - arithmetic, SQL, 196–197
  - collections, 413–416
  - comparison, 209–211
  - nested queries, 209–211
  - defined, 17
  - grouping, 415–416
  - logical comparison, SQL, 188–190
  - OQL collections, 413–416
  - spacial, 990–991
  - SQL query recovery, 188–190, 196–197, 209–211
  - SQL query translation into, 657–660
- Optical drives, 544
- Optimistic protocols, 781
- Optional field, records, 561–562
- OR logical connective, SQL, 209–210
- OR operator, *see* AND/OR/NOT operators
- Oracle
  - adaptive optimization, 735
  - array processing, 735–736
  - global query optimizer, 734–735
  - hints, 736
  - key-value store, 899
  - label-based security policy, 1155–1158
  - outlines, 736
  - physical optimizer, 733–734
  - query optimization in, 733–737
  - SQL plan management, 736–737
  - virtual private database (VPD) technology, 1156
- ORDBMS (object-relational database management system), 364
- ORDER BY clause
  - SQL, 197–198
  - XQuery, 446
- Order preserving, hashing, 577
- Ordered (sorted) records, 568–572
- Ordering field, records, 568
- OUTER JOIN operations, 216, 262–264
- Outer query, 209
- OUTER UNION operation, 264–265
- Outlines, Oracle, 736
- Overflow (transaction) file, 571
- Overlapping entities, 115, 126
- PageRank ranking algorithm, 1051
- Parallel algorithms
  - aggregate operations for, 686
  - architectures for, 683–684
  - interquery parallelism, 687
  - intraquery parallelism, 687
  - join techniques, 685
  - operator-level parallelism, 684–686
  - partitioning strategies, 684
  - projection and duplicate elimination, 685
  - query processing using, 683–687
  - selection conditions, 685
  - set operations for, 686
  - sorting, 684
- Parallel database architecture, 683
- Parallel processing, 747
- Parameters
  - binding, 329, 333
  - disks, 1167–1169
  - JDBC statement parameters, 333
  - SQL/CLI statement parameters, 329
  - stored procedure type and mode, 336–337
- Parametric (naïve) end users, 16
- Parametric user interfaces, 42
- Parent nodes, tree structures, 617
- Parser, query processing, 655
- Partial categories, 122
- Partial key, 79, 479
- Partial specialization, 115, 126
- Participation constraints, 77–78
- Partition algorithm, 1081
- Partition-hash join, 559, 674–675, 719, 930–931
- Partition tolerance, DDBs, 845
- Partitioned hashing, 632
- Partitioning strategies
  - NOSQL, 886
  - parallel algorithms, 684
- Partitions
  - OQL, 415–416
  - grouping and, 219, 415–416
  - SQL query retrieval and, 219
- Path expressions
  - OQL, 410–412
  - SQL, 386
  - XPath for, 443–445
- Path separators (/ and //), XML, 443
- Patterns, substring matching in SQL, 195–197

- PEAR (PHP Extension and Application Repository), 353–354
- Performance, Big data technology and, 945
- Performance monitoring, 45
- Periodic updates, SQL views, 230
- Persistent data, storage of, 545
- Persistent objects, 365, 373–374
- Persistent storage, 19–20
- Persistent storage modules, 336
- Phantom records, concurrency control and, 806–807
- PHP (Hypertext processor)
  - arrays, 345–346, 348–350
  - built-in variables, 352–353
  - comments in, 345
  - connecting to a database, 353–355
  - data collection and records, 355–356
  - error checking, 355
  - Extension and Application Repository (PEAR), 353–354
  - functions, 350–352
  - here documents, 347–348
  - HTML and, 343–346
  - middle-tier Web server as, 344
  - numeric data types for, 348
  - placeholders, 356
  - predefined variables, 345–346
  - query retrieval, 356–357
  - query submission, 355
  - text strings in, 346, 347–348
  - use of, 343–345
  - variable names for, 346, 347
  - Web programming using, 343–359
- Phrase queries, 1036
- Physical clustering, mixed records, 583
- Physical data independence, 38
- Physical data models, 33–34
- Physical database design
  - data storage and, 546
  - indexing design decisions, 645–646
  - indexing structures, 601–652
  - job mix factors for, 643–645
  - multilevel indexes, 613–617
  - relational databases (RDBs) with, 643–646
  - single-level ordered indexes, 602–613
- Physical database file structures, 641. *See also* Indexes
- Physical design, data modeling, 62
- Physical index, 638–639
- Physical optimization, queries, 724
- Physical optimizer, Oracle, 733–734
- Pin count, buffer management, 558
- Pin-unpin bit, database recovery cache, 816
- Pipelined parallelism, 687
- Pipelining
  - combining operations using, 681–683
  - iterators for implementation of, 682–683
  - materialized evaluation and, 681
  - pipelined evaluation, 682
  - processing information, 1028–1029
  - query processing using, 681–683
- Placeholders, PHP, 356
- Plan caching, query optimization, 730
- Pointers
  - B-trees, 620, 623–624
  - file records, 563, 575–576
- Polymorphism (operator overloading)
  - binding and, 377
  - ODBs, 366, 377
- Populating (loading) databases, 35
- Positional iterator, SQLJ, 323
- Practical relational model, 177–206.
  - See also* SQL (Structured Query Language) system
- Precompiler
  - DML command extraction, 44
  - embedded SQL and, 311, 314
- Predefined variables, PHP, 345–346
- Predicate, relation schema and, 156
- Predicate-defined subclasses, 113, 126
- Prefix compression, string indexing, 640
- PreparedStatement objects, JDBC, 333
- Preprocessor, embedded SQL and, 311, 314
- Primary file organization, 546
- Primary indexes, 602, 603–606
- Primary keys
  - arbitrary designation of, 477
  - normal form based on, 483–495
  - relational data modeling, 159
  - SQL constraints, 186–187
  - XML specification, 441
- Primary storage, 542, 543
- Prime/nonprime attributes, 477
- Printer servers, client/server architecture, 47
- Privacy issues and preservation, 1153–1154
- Privileged software use, 19
- Privileges, granting and revoking in SQL, 202
- Probabilistic model, IR, 1033–1034
- Probabilistic topic modeling, IR, 1059–1061
- Program variables
  - embedded SQL, 314–315
  - file operations, 565–566
- Program-data independence, 12
- Programming, *see* Database programming; SQL programming
- Programming languages
  - DBMS, 38–40
  - declarative, 40
  - design for database programming, 312–313, 339
  - impedance mismatch, 312–313
  - Java, 321–325, 358
  - PHP (Hypertext processor), 343–359
  - QBE (Query-by-Example), 1171–1178
  - XML, 434, 436–447
- Programming model, MapReduce (MR), 918–921
- Program-operation independence, 12
- Project attributes, 189
- PROJECT operation
  - degree of relations, 244
  - duplicate elimination and, 244–245
  - query processing, algorithms for, 676–678
  - relational algebra using, 243–245
- Prolog language, deductive databases, 1000–1003
- Proof by contradiction, 507
- Proof-theoretic interpretation of rules, 1005
- Properties of decomposition
  - attribute preservation condition, 513
  - dependency preservation, 514–515
  - insufficiency of normal forms, 513–514
  - nonadditive (lossless) join, 515–517, 519–523
  - RDB design and, 504, 513–518
  - universal relations and, 513
- Protection, databases, 6
- Proximity queries, 1036
- Public key encryption, 1151–1152
- Pure distributed database architecture, 869–871
- QBE (Query-by-Example) language
  - aggregate functions in, 1175–1177
  - grouping, 1175–1177
  - modifying the database, 1177–1178
  - retrievals in, 1171–1175
- Qualified association, UML class diagrams, 88
- Qualifier conditions, XML, 443
- Quantifiers
  - domain relational calculus, 279
  - existential, 271, 274
  - queries using, 274–276
  - transformation of, 274
  - tuple relational calculus, 271, 274–276
  - universal, 271, 274–276
- Queries
  - buffering (caching) modules for, 20, 42
  - compiler, 43–44
  - complex retrieval, 207–225
  - constant nodes, 273
  - Datalog language, 1004, 1010–1012
  - defined, 6
  - indexes for, 20
  - indexing hints in, 641–642
  - information retrieval (IR) systems, 1035–1037
  - interactive interface, 43–44
  - join condition, 189, 191
  - keyword-based, 41
  - named specification, OQL, 412–413
  - nested, 209–212
  - nonrecursive evaluation, 1010–1012
  - object query language (OQL), 408–416
  - ODMG model for, 408–416
  - optimizer, 44
  - outer, 209
  - processing in databases, 20



Queries (*continued*)

- quantifiers for, 274–276
- recursive, 223
- relation nodes, 273
- relational algebra for, 265–268
- select-from-where structure, 188–190
- selection condition, 189
- select-project-join, 189–190, 273
- spatial, 991
- SQL retrieval, 187–198, 207–225
- temporal constructs, 984–986
- TSQL2 language for, 984–986
- tuple relational calculus for, 272–276
- XML languages for, 443–447

Query block, 657–658

Query decomposition, DDBMS, 863–865

Query execution

- aggregate functions for, 709
- cost components for, 711–712
- GROUP-BY view merging, 705–706
- incremental view maintenance, 707–710
- materialized views for, 707–710
- nested subqueries, 702–704
- query evaluation for, 701–702
- subquery (view) merging
  - transformation for, 704–706

Query graphs

- internal query representation by, 655
- notation, 692–694
- query optimization, 692–697
- tuple relational calculus, 273–274

Query modification, SQL views, 229–230

Query optimization

- cost estimation for, 657, 710–713, 716–717
- cost functions for, 714–715, 717–
- cost-based optimization, 710–712, 716, 726–728
- data warehouses, 731–733
- distributed databases (DDBs), 859–863
- dynamic programming, 716, 725–726
- execution plan, display of, 729
- heuristic rules for, 657, 692, 697–701
- histograms for, 713
- JOIN operation for, 717–726
- multirelation queries, 721–724
- operation size estimation, 729–730
- Oracle, 733–737
- physical optimization, 724
- plan caching, 730
- query execution and, 701–712
- query processing compared to, 655–657
- query trees and graphs for, 692–697
- SELECT operation for, 714
- semantic query optimization, 737–738
- star-transformation optimization, 731–733
- top-*k* results, 730
- transformation rules for relational algebra operations, 697–699

Query optimizer, 655

## Query processing strategies

- aggregate operation implementation, 678–679
- anti-join (AJ) operator for, 658–660
- distributed databases (DDBs), 859–863
- external sorting algorithms, 660–663
- importance of, 656–657
- JOIN operation implementation, 668–676, 679–681
- parallel algorithms for, 683–687
- pipelining to combine operations, 681–683
- PROJECT operation algorithm, 676–678
- query block for, 657–658
- query optimization compared to, 655–657
- SELECT operation algorithms, 663–668
- semi-join (SJ) operator for, 658–660
- set operation algorithm, 676–678
- SQL query translation, 657–660
- steps for, 655–656

Query results

- bound columns approach, 329
- cursor (iterator variable) for, 312, 317–320
- embedded SQL, 312, 317–320
- impedance mismatch and, 312
- iterators for, 323–325
- OQL, 410–412
- path expressions, 386, 410–412
- PHP, 356–357
- SQL/CLI processing, 329
- SQLJ processing of, 323–325

Query retrieval

- aggregate functions in, 216–219
- alias for, 192
- arithmetic operators for, 196–197
- asterisk (\*) uses, 193, 218
- attribute name qualification, 191
- Boolean (TRUE/FALSE) statements for, 212–214
- CASE clause for, 222–223
- clauses used in, 198–199
- comparison operators, 188–191, 195–197
- complex queries, 207–225
- EXISTS function for, 212–214
- explicit sets of values, 214–215
- FROM clause for, 188–189, 197, 232
- grouping, 216–222
- joined tables (relations), 215–216
- LIKE comparison operator, 195–196
- logical comparison operators for, 188–190
- multiset of tuples, 188, 193–195
- nested queries, 209–212
- NULL values and, 208–209
- ORDER BY clause for, 197–198
- ordering results, 197
- PHP, 356–357
- QBE (Query-by-Example) language, 1171–1175
- recursive queries, 223

- renaming attributes, 192, 214–215
- SELECT statement (clause) for, 187–188, 194–195, 197
- select-from-where block, 188–191
- set operations for, 194–195
- set/multiset comparisons, 209–211
- SQL, 187–198, 207–225, 230–231
- substrng pattern matching, 195–197
- table set relations, 193–195
- three-valued logic for comparisons, 208–209
- tuple variables for, 192, 209–211
- UNIQUE function for, 212–214
- views (virtual tables) for, 230–231
- WHERE clause for, 188, 192–193, 197
- WITH clause for, 222–223

Query server, two-tier client/server architecture, 49

Query submission, PHP, 355

Query tree

- defined, 257
- heuristic optimization of, 694–694
- internal query representation by, 655
- notation, 257–259, 692–694
- query optimization, 692–697
- RDBMS use of, 257–259
- semantic equivalence of, 694–695

Query validation, 655

Question answering (QA) systems, 1061–1063

RAID (redundant arrays of inexpensive disks) technology

- bit-level striping, 584, 586
- block-level striping, 584–585, 586
- data striping, 584–585
- levels, 586–588
- mirroring, (shadowing), 585
- parallelizing disk access using, 542, 584–588
- performance, improvement with, 586
- reliability, improvement with, 585–586

Random-access memory (RAM), 543

Random access storage devices, 554

Range partitioning, 684, 886

Range relations, tuple variables and, 269–270

RDBMS (Relational database management system)

- query tree notation, 257–259
- two-tier client/server architecture and, 49

RDBs (relational databases)

- application flexibility with, 24
- data abstraction in, 24
- indexing for, 643–646
- integrity constraints and, 160–163
- physical database design in, 643–646
- relation schema sets as, 160
- schemas, 160–163
- temporal databases incorporating time in, 977–982

- tuple versioning, 977–982
- valid and invalid relational states, 160–161
- Reachability, object persistence and, 373–374
- Read/write head, disk devices, 551
- Read/write transactions, 748
- Real-time database technology, 4
- Reasoning mechanisms, 129
- Recall and precision metrics, IR, 1044–1046
- Record type (format), 560
- Record-at-a-time, file operations, 566
- Record-at-time DML, 40
- Record-based data models, 33
- Records
  - blocking, 563–564
  - data types, 560–561
  - data values, 560
  - fields, 560, 561–563, 568–569, 582–583
  - file storage, 560–564, 567–572, 582–583
  - fixed-length, 561–563
  - mixed, 582–583
  - ordered (sorted), 568–572
  - spanned versus unspanned, 563–564
  - unordered (heaps), 567–568
  - variable-length, 561–563
- Recoverability basis of schedules, 761–762
- Recoverable/nonrecoverable schedule, 761
- Recursive closure operations, 262
- Recursive queries, 223
- Recursive (self-referencing) relationships, 75
- Redis key-value cache, 900
- Redundancy control, 18–19
- REF keyword, 383, 386
- Reference types, OIDs created using, 383
- References
  - dot notation for path expressions, 386
  - inverse, 366, 370, 396–397
  - object identity from, 370
  - object type relationships, 369–370
  - relationships specified by, 386
  - SQL, 370, 386
- Referential integrity
  - constraints, 21, 163–165, 186–187
  - NULL values and, 163–164
  - relational data modeling, 163–165
  - SQL constraints, 186–187
- Referential triggered action clause, SQL, 186
- Reflexive association, UML class diagrams, 87
- Regression, data mining, 1091–1092
- Regression function, data mining, 1092
- Relation extension/intension, 152
- Relation nodes, query graphs, 273
- Relation schema
  - anomalies and, 465–467
  - assertion, 156
  - attribute clarity and, 464
  - degree (arity) of attributes, 152
  - facts, 156
  - functional dependency of, 471–474
  - goodness of, 459
  - interpretation of, 156
  - key of, 159
  - nested relations, 479–480
  - normalization of relations, 474–476
  - NULL value in tuples, 467–468
  - predicate, 156
  - redundant information in tuples, 465–467
  - relational database (RDB) design guidelines, 461–471
  - relational model constraints and, 157–165
  - relational model domains and, 152
  - semantics of, 461–465
  - spurious tuple generation, 468–471
  - superkey of, 158–159
  - universal, 471–474
- Relation state
  - current, 153
  - relational model domains and, 152–153
  - relational database, 160–161
  - tuple values in, 152–156
  - valid and not valid, 160–161
- Relational algebra
  - aggregate functions, 240, 260–261
  - binary operations, 240, 251–259, 262–264
  - expressions for, 239, 241–242, 245
  - formal relational modeling and, 239–240
  - generalized projection operation, 259–260
  - groupings, 260–261
  - operations, purpose and notation of, 258
  - procedural order of, 268
  - queries in, 265–268
  - query optimization and, 697–699
  - recursive closure operations, 262
  - set theory and, 246–251, 264–265
  - SQL query translation into, 657–660
  - transformation rules for operations, 697–699
  - unary operations, 240, 241–246
- Relational calculus
  - declarative expressions for, 268
  - domains and, 268, 277–279
  - formal relational modeling and, 240–241
  - nonprocedural language of, 268
  - query graphs, 273–274
  - relationally complete language of, 268
  - tuples and, 268–277
- Relational data models
  - attributes, 152–153
  - breaking cycle for tree-structure model conversion, 452–453
  - concepts, 150–157
  - constraints, 157–167
  - DBMS criteria and, 51–52
  - delete operation, 166, 167–168
  - domains, 151–152
  - entity integrity, 163–165
  - extraction of XML documents using, 447–449
  - flat files, 150
  - formal languages for, *see* Relational algebra; Relational calculus
  - insert operation, 166–167
  - key constraints, 21, 158–160, 163–165
  - mathematical relation of, 149
  - notation for, 156–157
  - operations, 165–168
  - referential integrity, 163–165
  - practical language for, *see* SQL (Structured Query Language)
  - relations, 152–156
  - representational model type, 33
  - retrievals (operations), 165–166
  - schemas, 152–165
  - table of values, 150–151
  - transactions, 169
  - tuples, 152–156
  - update (modify) operation, 166, 168–169
- Relational database (RDB) design
  - algorithms for schema design, 519–523, 524–527
  - bottom-up method, 460, 504
  - by analysis, 503
  - by synthesis, 504, 503
  - dangling tuple problems, 523–524
  - data model mapping for, 289
  - designer intention for, 459–460
  - EER-to-relational mapping, 298–303
  - ER-to-relational mapping, 290–298
  - functional dependency and, 471–474, 505–512, 527–528, 532
  - implementation (physical storage) level, 459–460
  - inclusion dependency and, 531–532
  - inference rules for, 505–509, 527–528
  - join dependency (JD) and, 494–495, 530–531
  - keys for, 474–483
  - logical (conceptual) level, 459–460
  - multivalued dependency (MVD) and, 491–494, 527–530
  - normal forms, 474–495, 513–514, 528–533
  - normalization algorithm problems, 524–527
  - normalization of relations, 474–476, 482, 485, 486–487, 493–494
  - NULL value problems, 523–524
  - ODBs compared to, 405–406
  - properties of decomposition, 504, 513–518
  - relation schema, guidelines for, 461–471
  - top-down method, 460
  - universal relations, 471–474, 504

- Relational database management system, *see* RDBMS (Relational database management system)
- Relational database state, 160–161
- Relational databases, *see* RDBs (relational databases)
- Relational operators for deductive databases, 1010
- Relationally complete language of, 268
- Relationships
  - aggregation, 87–88
  - associations, 87–88
  - attributes of, 78
  - attributes, as, 74
  - binary types, 76–78, 293–295
  - cardinality ratios for, 76–77
  - comparison of ternary and binary, 88–91
  - conceptual data models, 33
  - constraints on, 76–78, 91–92
  - degree of types, 71–74, 88
  - entity participation in, 72–73
  - ER models and, 72–78, 88–92
  - ER-to-relational mapping, 293–296
  - existence dependency, 77–78
  - identifying, 79
  - instances, 72
  - inverse, 396–397
  - multivalued attributes, 295–296
  - n*-degree, 88–92, 296
  - ODMG model objects, 396–397
  - order of instances in, 87
  - participation constraints of, 77–78
  - recursive (self-referencing), 75
  - role names and, 75
  - sets, 72
  - structural constraints of, 78
  - subtype/supertype, 375–376
  - ternary, 88–92
  - type, 72–78, 126
  - type hierarchies, 375–376
  - UML class diagrams, 87–88
- Reliability, DDBs, 844–845
- RENAME operator (*p*), 245–246
- Renaming attributes in SQL, 192, 214–215
- Repeating field, records, 561–563
- Replication models, 886
- Replication transparency
  - DDBs, 843
  - NOSQL, 885–886, 894
- Representational (implementation) data models, 33
- Resource Description Framework (RDF), 447
- ResourceManager (RM), YARN, 941–942
- RESTRICT option, SQL, 233, 234
- Result equivalence, schedules, 765
- ResultSet object JDBC, 334–335
- Retrieval operations
  - files, 564–565
  - object information, 371
  - relational data models, 165–166
  - selection conditions, 564–565
- Retrieval, 1027
- RETURN clause, XQuery, 446
- ROLAP (relational OLAP) function, 1114
- Role-based access control (RBAC), 1121, 1137–1139
- Role names, 75
- Roles of domain attributes, 152
- Root, tree structures, 617
- Root element, XML, 440
- Root tag, XML documents, 434
- Rotational delay (latency), disk devices, 552
- Round-robin partitioning, 684
- Row, SQL, 179
- Row-based constraints, SQL, 187
- Row-level access control, 1139–1140
- ROW TYPE command, 380
- RSA public key encryption algorithm, 1152
- Rules
  - active databases systems, 22
  - active rules, 962–964, 970–973
  - association rules, 1073–1084
  - axioms, 1005
  - deductive database systems, 22
  - deductive databases, 1000, 1005–1007
  - defined, 1000
  - force/no-force rules, 817–818
  - 4NF schema, 527–528
  - functional dependencies, 505–509, 527–528
  - inference rules, 505–509, 527–528
  - inferencing information using, 22
  - interpretation of, 1005–1007
  - models for, 1005–1006
  - model-theoretic interpretation of, 1005
  - proof-theoretic interpretation of, 1005
  - stored procedure for, 22
  - theorem proving, 1005
  - triggers as, 22
- Runtime, MapReduce (MR), 927–930
- Runtime database processor, 44, 655
- Safe expressions, 276–277
- Sampling algorithm, 1076–1077
- Scalability
  - DDBs, 845
  - NOSQL, 885
- Scale-invariant feature transform (SIFT), 998
- Scanner, query processing, 655
- Schedules (histories)
  - cascading rollback phenomenon, 762
  - committed projection of, 760
  - complete schedule conditions, 760
  - concurrency control and serializability, 770–771
  - conflict equivalence of, 765–766
  - conflicting operations in, 759–760
  - debt–credit transactions, 773
  - nonserial schedules, 763, 764–765
  - operation semantics for, 773
  - recoverability basis of, 761–762
- recoverable/nonrecoverable schedule, 761
- result equivalence of, 765
- serial schedules, 763–764
- serializability basis of, 763–766
- serializable schedules, 763, 765–766
- strict schedule, 762
- testing for serializability, 767–770
- transaction processing, 759–773
- transactions for, 759–760
- view equivalence, 771–772
- view serializability, 771–772
- Schema-based (explicit) constraints, 157
- Schema change statements
  - ALTER command, 233–234
  - DROP command, 233
  - schema evolution command use, 232–233
- Schema diagram, 34–35
- Schema matching, 1052
- Schemaless documents, XML, 432–433
- Schemas
  - authorization identifier, 179
  - bottom-up conceptual synthesis, 119
  - catalog collection of, 35, 38, 180
  - conceptual level, 37, 61–62
  - constraints and, 157–165
  - constructs, 35
  - data independence and, 37–38
  - database descriptions, 34
  - database state (snapshot) and, 35
  - database requirements, 122–124
  - descriptors, 179
  - design creation (conceptual) of, 61–62
  - EER modeling and, 119–120, 122–124
  - EER schema to ODB schema, 407–408
  - ER diagram notation for, 81, 83–85
  - ER modeling and, 61–62
  - evolution, 35
  - external level (views), 37
  - intention, 35
  - interface inheritance, ODL, 404–405
  - internal level, 36
  - mappings, 37, 407–408
  - meta-data storage of, 35
  - naming constructs, 82
  - ODB conceptual design and, 407–408
  - ODL, 400–403
  - refinement using generalization and specialization, 119–120
  - relation, 157–160, 163–165
  - relational database, 160–163
  - SQL concepts, 179–180
  - three-schema architecture, 36–38
  - top-down conceptual refinement, 119
  - XML language, 434, 436–441
- Script functions, HTML, 428
- Search, B-trees, 625–626
- Search engines
  - desktop, 1025
  - Lucern, 1043–1044
  - Web search, 1047

- Search relevance, IR, 1044–1047
- Search techniques
  - conjunctive selection, 665–666
  - disjunctive selection, 666–667
  - keyword-based, 41
  - query processing, 663–667
  - SELECT operation algorithms, 663–667
  - simple selection, 663–665
  - Web database applications, 4
- Search trees, dynamic multilevel indexes, 618–619
- Second normal form (2NF)
  - definition of, 481
  - full functional dependency and, 481–482
  - general definition of, 484–486
  - normalizing relations, 482, 484–486
  - primary key and, 483–484
- Secondary access path, indexing, 601
- Secondary indexes, 603, 609–612
- Secondary storage
  - capacity of, 534
  - devices for, 547–556
  - random access devices, 554
  - sequential access devices, 554–555
  - solid-state drive (SSD), 542
- Security, *see* Data security; Database security
- Security and authorization subsystems, 19
- Seek time, disk devices, 552
- SELECT clause statement
  - ALL option with, 194–195
  - AS option with, 196
  - DISTINCT option with, 188, 194
  - mandatory use of, 197
  - multiset tables and, 194–195
  - SQL query retrieval and, 187–188, 194–197
- SELECT operation
  - Boolean expressions (clauses), 241–242
  - cascade (sequence) with, 243
  - conjunctive selection, 665–666
  - cost functions for, 714
  - degree of relations, 243
  - disjunctive selection, 666–667
  - estimating selectivity of conditions, 667–668
  - implementation options for, 663
  - query processing algorithms, 663–668
  - relational algebra using, 241–243
  - search methods for, 663–667
  - selectivity of a condition, 243, 667–668
  - simple selection, 663–665
- SELECT operator ( $\sigma$ ), 241
- Select...from...where structure, OQL, 409
- Select-from-where block, SQL, 188–191
- Select-project-join query, 189–190, 273
- Selection conditions
  - domain variables, 278
  - file operations, 564–565
  - parallel algorithms, 685
  - WHERE clause queries, 189
- Selective inheritance, 377
- Selectivity
  - join operations, 254, 719–720
  - of a condition, 243, 667–668
- Self-describing data, 10, 427
- Self-describing data models, 34
- Self-describing documents, 425. *See also* JSON; XML (EXtended Markup Language)
- Semantic approach, IR, 1028
- Semantic data models
  - abstraction concepts, 129–133
  - EER modeling, 107–108
  - ontology for, 132–134
- Semantic equivalence, query trees, 694–695
- Semantic heterogeneity, 857–858
- Semantic model, IR, 1034–1035
- Semantic query optimization, 737–738
- Semantic tagging, images, 998–999
- Semantics
  - attribute clarity, 461–465
  - data constraints, 21
  - functional dependency of, 472–473
  - relation schema, 461–465
  - RDB design, 461–465, 472–473
  - schedule operations, 773
- Semi-join (SJ) operator, 658–660, 681, 719–720, 862–863
- Semistructured data, XML, 426–428
- Separator characters, records, 561
- Sequence of interaction, database programming and, 313–314
- Sequence of operations, relational algebra, 245–246
- Sequential access storage devices, 554–555
- Sequential pattern discovery, data mining, 1091
- Serial ATA (SATA), 551
- Serial schedules, 763–764
- Serializability
  - basis of schedules, 763–766
  - concurrency control and, 770–771
  - testing for, 767–770
- Serializable schedules, 763, 765–766
- Server, defined, 48
- Servers
  - application, 44
  - database, 44
  - DBMS module, 31
- SET clause, SQL, 201
- SET CONNECTION command, SQL, 316
- Set constructor, 369
- SET DIFFERENCE operation, 247–249
- Set operations
  - anti-join (AJ) operator for set difference, 677–678
  - parallel algorithms, 686
  - query processing, algorithms for, 676–678
  - SQL, 194–195
- Set theory
  - CARTESIAN PRODUCT operation, 249–251
  - INTERSECTION operation, 247–249
  - MINUS operation, 247–249
  - OUTER UNION operation, 264–265
  - relational algebra operations from, 246–251, 264–265
  - SET DIFFERENCE operation, 247–249
  - type compatibility, 247
  - UNION operation, 246–249
- Set type, legacy data modeling with, 53
- Set-at-a-time, file operations, 566
- Set-at-time DML, 40
- Sets
  - explicit set of values, 214
  - multiset comparisons, SQL query retrieval, 209–211
  - parentheses for, 214
  - SQL table relations, 188, 193–195
- Shadow directory, 826
- Shadow paging, database recovery, 826–827
- Shadowing, 816
- Sharding
  - DDBs, 847–848
  - NOSQL, 886, 894–895
- Shared-disk architecture, 683
- Shared/exclusive (read/write) locks, 784–786
- Shared-memory architecture, 683
- Shared-nothing architecture, 684
- Shared subclasses, 118, 301
- Shared variables in embedded SQL, 314
- Signature of operations, 366, 397. *See also* Interfaces
- Simple (atomic) attributes, 65–66
- Simple elements, XML, 431
- Simple Object Access Protocol (SOAP), 447
- Simple selection, search methods for, 663–665
- Single character replacement symbol ( $\_$ ), 195–196
- Single inheritance, 118–119
- Single-level ordered indexes
  - clustering indexes, 602, 606–608
  - concept of, 602–603
  - physical database design and, 602–613
  - primary indexes, 602, 603–606
  - secondary indexes, 603, 609–612
- Single-relation options, EER-to-relational mapping, 299–300
- Single-sided disks, 547
- Single time point, 976
- Single-user DBMS systems, 51
- Single-valued attribute, ER modeling, 66

- Small computer system interface (SCSI), 551
- Snapshot isolation
  - concurrency control and, 758, 781, 799–800
  - defined, 775
  - SQL transaction support and, 775–776
- Snapshot (database) state, 35
- Snowflake schema, 1108–1109
- Social search, IR, 1058–1059
- Software engineers, 16
- Solid-state device (SSD) storage, 553–555
- Solid-state drive (SSD), secondary storage of, 542
- Sophisticated end users, 16
- Sorting phase, external algorithms, 661
- Sort-merge join, 559, 719, 930
- Spanned versus unspanned records, 563–564
- Spatial analysis operations, 988
- Spatial colocation rules, 993–994
- Spatial databases
  - analytical operations, 988
  - applications of spatial data, 994
  - data mining, 993–994
  - data types, 989–990
  - enhanced data models, 962, 987–994
  - indexing, 991–993
  - models of information, 990
  - object storage by, 987–988
  - operators, 990–991
  - queries, 991
- Specialization
  - attribute-defined, 114
  - conceptual schema refinement, 119–120
  - constraints on, 113–116
  - defined, 110
  - design choices for, 124–128
  - disjointness (d notation), 114–115
  - EER diagram notation for, 109, 110
  - EER modeling concept, 108, 110–120, 124–128
  - EER-to-relational mapping options, 298–301
  - hierarchies, 116–119
  - instances of, 111–112
  - lattices, 116–119
  - partial, 115
  - semantic modeling process, 131
  - total, 115
  - UML notation for, 127–128
- Specialized servers, client/server architecture, 47
- Specification, ontology and, 134
- Speech input and output, 41
- Spurious tuple generation, RDB design and, 468–471
- SQL (Structured Query Language) system
  - active database techniques, 202
  - arithmetic operators, 196–197
  - assertions, 158, 156, 165, 225–226
  - attribute data types in, 182–184
  - catalog concepts, 179–180
  - CHECK clause, 187
  - comparison operators, 188–191, 195–197
  - complex queries, 207–225
  - constraints, 165, 184–187, 225–227
  - core specifications, 178
  - CREATE ASSERTION statement, 225–226
  - CREATE TABLE command, 180–182
  - CREATE TRIGGER statement, 225, 226–227
  - data definition, 179
  - DBMS use of, 177–178
  - DELETE command, 200
  - domains, 184
  - encapsulation of operations, 384–385
  - extensions, 178
  - function overloading, 385
  - granting and revoking privileges, 202
  - history of, 178
  - index creation, 201–202
  - inheritance, type specification of, 385
  - INSERT command, 198–200
  - logical comparison operators, 188–190
  - NOSQL database system and, 26
  - object identifiers, 383
  - object-relational systems, 202
  - ODB extensions to, 379–386
  - operators, query translation into, 657–660
  - practical relational model, 177–206
  - query processing, translation for, 657–660
  - query retrieval, 187–198, 207–225
  - reference types, 383
  - relational algebra, query translation into, 657–660
  - relational data models and, 51, 165
  - schema change statements, 232–234
  - schema concepts, 179–180
  - syntax of, 235
  - table creation, 383–384
  - transaction support, 773–776
  - triggers, 158, 165, 226–227
  - UPDATE command, 200–201
  - user-defined types (UDTs), 380–384
  - views (virtual tables), 228–232
  - XML data creation functions (XML/SQL), 453–455
- SQL injection
  - bind variables, 1145–1146
  - code injection, 1144
  - database security, 1143–1146
  - filtering input, 1146
  - function call injection, 1144–1145
  - function security for, 1146
  - manipulation, 1143–1144
  - protection against attacks, 1145–1146
  - risks associated with, 1145
- SQL plan management, Oracle, 736–737
- SQL programming
  - comparison of approaches, 338–339
  - database programming language approaches, 309–314, 339
  - database stored procedures, 335–338
  - dynamic SQL, 320–321
  - embedded SQL, 311, 314–320, 338–339
  - JDBC: SQL class library, 331–335
  - library of functions or classes for, 311–312, 326–335, 339
  - query specification and, 320–321
  - SQL/CLI (SQI call level interface), 326–331
  - SQLJ: Java commands, 321–325
- SQL server, two-tier client/server architecture, 49
- SQL/CLI (SQI call level interface)
  - connection record, 327–328
  - database programming with, 326–331
  - description record, 327–328
  - environment record, 327–328
  - handles for records, 328
  - statement record, 327–328
  - steps for programming, 328–331
- SQL/PSM (SQL/persistent stored modules), 337–338
- SQLCODE variable, 316
- SQLJ
  - embedding SQL commands in Java, 321–325
  - exceptions for error handling, 322–323
  - iterators for, 323–325
  - query result processing, 323–325
- SQLSTATE variable, 316
- Standalone users, 16
- Standards, enforcement of, 22
- Star schema, 1108
- STARBURST, statement-level rules in, 970–972
- Star-transformation optimization, 731–733
- Starvation, 792
- State constraints, 165
- State of an object or literal, 387
- Statement object JDBC, 335
- Statement parameter
  - binding, 329, 333
  - JDBC, 333
  - SQL/CLI, 329
- Statement record, SQL/CLI, 327–329
- Statement string, SQL/CLI, 329
- Statement-level rules, STARBURST, 970–972
- Statement-level trigger, 967
- Static files, 566
- Static hashing, 577
- Statistical approach, IR, 1028
- Statistical database security, 1146–1147
- Steal/no-steal rules, 817–818
- Stemming, IR text processing, 1038



- Stopword removal, IR text processing, 1037–1038
- Storage
  - architectures for, 588–592
  - automated storage tiering (AST), 591
  - big data, 3
  - buffering blocks, 541, 556–560
  - capacity, 543
  - cloud, 3
  - column-based, indexing for, 642
  - database catalog for, 10–11
  - database organization of, 545–546
  - database reorganization, 45
  - devices for, 543–545, 547–556
  - Fibre Channel over Ethernet (FCoE), 590–591
  - Fibre Channel over IP (FCIP), 590
  - file records, 560–564, 567–572, 582–583
  - files, 10–11, 560–572, 582–583
  - hashing techniques, 572–582
  - Internet SCSI (iSCSI), 590
  - memory hierarchies, 543–545
  - meta-data, 6, 10
  - network-attached storage (NAS), 589–590
  - object-based, 591–592
  - objects, 987–988
  - persistent, 19–20, 545
  - primary, 542, 543
  - program objects, 19–20
  - RAID technology, 542, 584–588
  - secondary, 542, 543, 547–556
  - spatial databases for, 987–988
  - storage area networks (SANs), 588–589
  - tertiary, 542, 543
  - XML documents, 442–443
- Storage area networks (SANs), 588–589
- Storage definition language (SDL), 39
- Storage devices
  - databases, organization and, 545–546
  - disks, 547–553
  - flash memory, 543–544
  - magnetic tape, 544–545, 555–556
  - memory, 543–545, 547–556
  - optical drives, 544
  - secondary, 547–556
  - solid-state device (SSD), 553–555
- Stored attribute, 66
- Stored data manager, 42, 44
- Stored procedures
  - CALL statement, 337
  - database programming and, 335–338
  - parameter type and mode, 336–337
  - persistent storage modules, 336
  - rule enforcement using, 22
  - SQL/PSM (SQL/persistent stored modules), 337–338
- Stream-based processing, 682. *See also* Pipelining
- Strict schedule, 762
- Strings. *See also* Text strings
  - character data types, 182–183
  - double quotations ( " ") for, 196, 347
  - indexing, 640
  - prefix compression, 640
  - single quotations ( ' ') for, 182, 196, 347
  - SQL use of, 182–183, 195–197
  - substring pattern matching, 195–197
- Strong entity types, 79
- Struct (tuple) constructor, 368, 369
- Structural constraints, 78
- Structured data, XML, 426
- Structured data extraction, WEB, 1052
- Structured objects and literals, 388, 396
- Structured Query Language, *see* SQL (Structured Query Language)
- Subclasses
  - class relationships, 108–110
  - defined, 126
  - defining predicate of, 113–114
  - EER diagram notation for, 109
  - EER modeling concept, 108–110, 126
  - EER-to-relational mapping, 301
  - entity type as, 110
  - inheritance, 110, 117–119, 301
  - IS-A relationship, 109, 126
  - leaf class (UML node), 127
  - local attributes of, 110–111
  - overlapping entities, 115
  - predicate-defined, 113–114
  - shared, 118, 301
  - specialization of set of, 110–112
  - specific relationship types, 110–111
  - union type, 108, 120–122
  - user-defined, 114
- Subqueries
  - nested, 702–704
  - query optimization and, 702–706
  - unnesting (decorrelation), 704
  - view merging transformation, 704–706
- Substring pattern matching, SQL, 195–197
- Subtrees, 617
- Subtypes, 375–376
- SUM function
  - grouping, 260
  - SQL, 217
- Superclasses
  - base class (UML root), 127
  - categories of, 120–122
  - class relationships, 109
  - EER modeling concept, 109, 110, 126
  - entity type as, 110
  - inheritance, 110, 117–118
  - subclass relationships, 110, 117–118
- Superkey, 158–159, 476–477
- Supertypes, 375–376
- Surrogate key, 302
- Symmetric key algorithms, 1150–1151
- Synthesis, RDB design by, 503, 504
- System analysts, 16
- System designers and implementers, 17
- System log
  - database recovery, 814, 817, 818–819
  - modifications for database security, 1125
  - transaction processing, 755–756
- Table inheritance, SQL, 385
- Table of values, 150–151
- Table-based constraints, SQL, 184–187
- Tables
  - ALTER TABLE command, 180
  - base relations, 180, 182
  - CREATE TABLE command, 180–182
  - data definition statements, 180–182
  - database recovery, 828–831
  - inner join, 215–216
  - joined relations, 215–216
  - multiset operations, 193–195
  - multiway join, 216
  - NATURAL JOIN operation, 215
  - OUTER JOIN operations, 216
  - query retrieval and, 193–195
  - query retrieval and, 193–195, 215–216
  - sets of relations in, 188, 193–195
  - transaction, 828–831
  - trigger activation from, 22
  - UDT creation of for SQL, 383–384
  - views (virtual tables), 228–232
  - virtual relations, 82
- Tags
  - attributes, 430
  - document body specification, 429
  - document header specifications, 428
  - end/start tag (</...>), 428
  - HTML tag (<...>), 428
  - mark up of documents using, 428–429
  - notation and use, HTML, 428–430
  - semantic tagging of images, 998–999
  - XML unstructured data and, 428–430
- Tape jukeboxes, 544
- Taxonomy, ontology as, 134
- Temporal databases
  - applications of, 974
  - calendar, 975
  - enhanced data models, 962, 974–987
  - implementation considerations, 982
  - incorporating time, 977–984
  - object-oriented databases for, 982–984
  - relational databases for, 977–982
  - time representation, 975–977
  - versioning, 977–984
- Temporal querying constructs, 984–986
- Temporary update problem, transaction processing, 750
- Ternary relationships
  - binary relationships compared to, 88–89
  - degree of, 73–74
  - ER diagrams, 88–92
  - notation for diagrams, 88–89
- Tertiary storage, 542, 543

- Testing for serializability, 767–770
- Text/document source, multimedia
  - databases, 996
- Text preprocessing
  - information extraction (IE), 1040
  - information retrieval (IR), 1037–1040
  - stemming, 1038
  - stopword removal, 1037–1038
  - thesaurus use, 1038–1039
- Text strings
  - double-quoted, 347–348
  - interpolating variables within, 347
  - length of, 346
  - PHP programming, 346, 347–348
  - single-quoted, 347–348
- Thematic search, 989
- Theorem proving, 1005
- Thesaurus
  - IR text processing, 1038–1039
  - ontology as, 134
- THETA JOIN condition, 252
- Third normal form (3NF)
  - algorithm for RDB schema design, 519–522
  - definition of, 483
  - dependency preservation and, 519–522
  - general definition of, 486–487
  - nonadditive (lossless) join
    - decomposition and, 519–522
  - normalizing relations, 485, 486–487
  - primary key and, 483–484
  - transitive dependency and, 483
- Thomas's write rule, 795
- Three-schema architecture, 36–38
- Three-tier/client-server architecture
  - discrete databases (DDBs), 872–875
  - Web applications, 49–51
- Three-valued logic for SQL NULL
  - comparisons, 208–209
- Thrown exceptions, SQLJ, 322–323
- TIME data type, 183
- Time period, 976
- Time reduction, development of, 22–23
- Time representation, temporal databases, 975–977
- Time series data, 986–987
- Time series management systems, 987
- Timeouts, deadlock prevention, 792
- TIMESTAMP data type, 183–184
- Timestamp ordering (TO)
  - algorithm, 793
  - basic, 794
  - concurrency control based on, 792–795
  - multiversion technique based on, 796
  - strict, 794–795
  - Thomas's write rule for, 795
- Timestamps
  - concurrency control and, 781, 790–791, 793
  - deadlock prevention using, 790–791, 793
  - generation of, 793
  - transaction timestamps, 790–791
- Tool developers, 17
- Tools, DBMS, 45–46
- Top-down conceptual refinement, 119
- Top-down method, RDB design, 460
- Top-*k* results, query optimization, 730
- Topological relationships, 989
- Total categories, 122
- Total specialization, 115, 126
- Transaction management, DDBs, 857–859
- Transaction processing
  - commit point, 756
  - concurrency control, 749–752
  - concurrency of, 746–747
  - data buffers, 748–749
  - database items, 748
  - DBMS-specific buffer replacement
    - policies, 756–757
  - read/write transactions, 748
  - recovery for, 752–753
  - schedules (histories), 759–773
  - single-user versus multiuser systems, 746–747
  - SQL transaction support, 773–776
  - system log, 755–756
  - systems, 745
  - transaction failures, 752–753
  - transaction states, 753–754
  - transactions for, 747–749, 757–758
- Transaction rollback, database recovery, 819
- Transaction server, two-tier client/server
  - architecture, 49
- Transaction tables, database recovery, 828–831
- Transaction time dimensions, 976–977
- Transaction time relations, 979–980
- Transaction timestamps, deadlock
  - prevention, 790–791
- Transaction-id, 755
- Transactions
  - atomicity property, 14, 757
  - certification of, 781
  - concurrency control and, 781, 798–799, 807
  - consistency preservation, 757
  - database recovery, 821
  - debt–credit, 773
  - defined, 6, 169
  - desirable properties of, 757–758
  - durability (permanency) property, 758
  - interactive, 807
  - isolation property, 14, 758
  - multiuser processing, 13–14
  - not affecting database, 821
  - OTLP systems, 14, 52, 169
  - relational data modeling, 169
  - user-defined functional requirements, 61
  - validation (optimistic) of, 781, 798–799
- Transient data, storage of, 545
- Transient objects, 365, 373
- Transition constraints, 165
- Transitive dependency, 3NF, 483
- Transparency, DDBs, 843–844
- Tree search data structures, *see* B-trees; B<sup>+</sup>-trees
- Tree-structured data models
  - attributes, 433
  - breaking graph cycles for conversion
    - to, 452–453
  - data-centric documents, 431
  - data mining, 1077–1080, 1085–1086
  - decision trees, 1085–1086
  - document-centric documents, 431
  - document extraction using, 447–453
  - elements, 430–431
  - frequent-pattern (FP) tree, 1077–1080
  - graph conversion into, 452–453
  - hierarchies for, 116, 452–453
  - hybrid documents, 431
  - schemaless documents, 432–433
  - XML, 51, 430–433, 447–453
- Triggers
  - active databases, 963–967, 973–974
  - database tables and, 22
  - CREATE TRIGGER statement, 225, 226–227
  - database monitoring, 226–227
  - event-condition-action (ECA)
    - components, 227, 963–964
  - Oracle notation for, 965–967
  - SQL, 158, 165, 226–227
  - SQL-99 standards for, 973–974
- Trivial/nontrivial MVD, 493
- Truth value of atoms, 270, 277
- TSQL2 language, 984–986
- Tuning indexes, 640–641
- Tuple relational calculus
  - expressions, 270–271, 276–277
  - formulas (conditions), 270–271
  - nonprocedural language of, 268
  - quantifiers, 271, 274–276
  - queries using, 272–276
  - query graphs, 273–274
  - range relations, 269–270
  - requested attributes, 269
  - safe expressions, 276–277
  - selected combinations for, 269
  - variables, 269–270
- Tuple variables
  - alias of attributes, 192
  - bound, 271
  - free, 271
  - iterators, 189
  - range relations and, 269–270
- Tuples
  - alternative definition of a relation and, 154–155
  - anomalies and, 465–467
  - asterisk (\*) for rows in query results, 218



- atomic value of, 155
- attribute ambiguity and, 191–192
- CHECK clause for, 187
- CROSS PRODUCT operation for
  - combinations, 192–193
- dangling tuple problems, 523–524
- delete operation for, 166, 167–168
- embedded SQL retrieval of, 311, 314–317
- grouping and, 219
- mapping relations with, 154
- matching, 264–265
- multisets of, 193–195
- nested query values, 209–211
- $n$ -tuple for relations, 152
- NULL value of, 155–156, 163, 467–468
- ordering of, 154–155
- OUTER UNION operation and, 264–265
- parentheses for comparisons, 210
- partially compatible relations, 264
- partitioning relations into, 219
- precompiler or preprocessor for
  - retrieval of, 311, 314
- query retrieval and, 191–195, 209–211
- RDB design problems, 523–524
- redundant information in, 465–467
- referential integrity of, 163
- relation schema for RDB design, 465–471
- relation state values, 152–156
- row-based constraints, 187
- separate groups for NULL grouping
  - attributes, 219
- set of, 154–155
- spurious tuple generation, 468–471
- SQL tables and, 187, 191–195
- type (union) compatibility, 247
- update (modify) operation for, 166, 168–169
- versioning, 977–982
- Two-phase locking (2PL)
  - basic 2PL, 788
  - concurrency control, 782–792, 796–797
  - conservative 2RL, 788
  - deadlock, 789–792
  - expanding (first) phase, 786
  - locks for, 782–786
  - multiversion concurrency control and, 796–797
  - protocol, 786–788
  - rigorous 2PL, 789
  - shrinking (second) phase, 786
  - starvation, 792
  - strict 2PL, 788–789
  - subsystem for, 789
- Two-tier client/server architecture, 49
- Two-way join, 668
- Type (class) hierarchies
  - constraints on extents corresponding to, 376–377
  - functions in, 374–375
  - inheritance, 385
  - ODBs, 366, 374–377
  - subtype/supertype relationships, 375–376
  - visible/hidden attributes, 371, 375
- Type (union) compatibility, 247
- Type constructors
  - array, 369
  - atom, 368, 369
  - bag, 369
  - collection (multivalued), 369
  - dictionary, 369
  - list, 369
  - object definition language (ODL)
    - and, 369
  - object operation, 371
  - ODB objects and literals, 368–370
  - references to object type relationships, 369–370
  - set, 369
  - SQL, 379
  - struct (tuple), 368, 369
  - type structures and, 368–370
- Type generators
  - ODB objects and literals, 368–369
  - ODMG models, 394–395
- Type inheritance, 385
- Type structures, 368–370. *See also* Type constructors
- UDTs (User-defined types)
  - arrays, 383
  - built-in functions for, 384
  - CARDINALITY function, 383
  - CREATE TYPE command, 380–383
  - dot notation for, 383
  - encapsulation of operations, 384–385
  - inheritance specification (NOT FINAL), 385
  - SQL, 380–385
  - table creation based on, 383–384
- UML (Unified Modeling Language)
  - aggregation, 87–88
  - associations, 87–88
  - base class, 127
  - bidirectional associations, 87
  - class diagrams, 85–88, 127–128
  - EER models and, 127–128
  - ER models and, 60, 85–88
  - leaf class, 127
  - links, 87
  - qualified association, 88
  - reflexive association, 87
  - unidirectional association, 87
- Unary operations
  - assignment operations ( $\leftarrow$ ) for, 245
  - Boolean expressions (clauses), 241–242
  - cascade (sequence) with, 243
  - defined, 243
  - degree of relations, 243, 244
  - duplicate elimination and, 244–245
  - PROJECT operation, 243–245
  - relational algebra and, 240, 241–246
  - renaming attributes, 245–246
  - SELECT operation, 241–243
  - selectivity of condition, 243
  - sequence of operations for, 245–246
- Unauthorized access restriction, 19
- UNDO/REDO algorithm, 815, 818
- Unidirectional association, UML class diagrams, 87
- Unified Modeling Language, *see* UML (Unified Modeling Language)
- UNION operations
  - matching tuples, 264–265
  - OUTER UNION operation, 264–265
  - partially compatible relations, 264
  - relational algebra, 264–265
  - SQL sets, 194–195
- Union types
  - categories of, 120–122, 302–303
  - EER diagram notation for, 120
  - EER modeling concept, 108, 120–122
  - EER-to-relational mapping, 302–303
  - set union operation ( $\cup$ ), 120
  - surrogate key for, 302
- UNIQUE function, SQL query retrieval, 212–214
- Unique keys, 160
- Uniqueness constraints
  - ER model entity types, 68–68
  - key attributes as, 68–69
  - key constraints with, 158–160
  - relation schema and, 158–160
- Universal quantifiers, 271, 274–276
- Universal relation assumption, 513
- Universal schema relations, 471–474, 504, 513
- Universe of Discourse (UoD), 5
- Unnest relation, 1NF, 479–480
- Unordered file records (heaps), 567–568
- Unrepeatable read problem, transaction processing, 752
- Unstructured data, XML, 428–430
- Unstructured information, 1022
- Unstructured/semistructured data
  - handling, Big data technology and, 945
- Update (modify) operations
  - relational data models, 166, 168–169
  - files, 564–565
  - relational data models, 166, 168–169
  - selection conditions for, 564–565
  - tuple modification using, 166, 168–169
- Update anomalies, RDB design and, 465–467
- UPDATE command, SQL, 200–201
- Update decomposition, DDBMS, 863–865
- Update strategies for SQL views, 230–232
- Upgrading locks, 786

- User views, 37
- User-defined subclass, 114, 126
- User-defined types, *see* UDTs (User-defined types)
- Utilities, DBMS functions, 45
- Valid documents, XML, 434
- Valid state, databases, 35, 160–161
- Valid time, temporal databases, 976
- Valid time relations, temporal databases, 977–979
- Validation (optimistic) concurrency control, 781, 798–799
- Value (state) of an object or literal, 387
- Value sets (domains) of attributes, 69–70
- Variable-length records, 561–563
- Variables
  - built-in, 352–353
  - communication, 316
  - domain, 277
  - embedded SQL, 314–316
  - iterator, OQL, 409–410
  - interpolating within text strings, 347
  - names for, 346, 347
  - PHP, 345–347, 352–353
  - predefined, 345–346
  - program, 314–315
  - shared, 314
  - tuple, 189, 192, 169–170
- Vector space model, IR, 1031–1033
- Versioning
  - attribute approach, 982–984
  - NOSQL, 887, 899, 900–902
  - object-oriented databases
    - incorporating time, 982–984
  - relational databases incorporating time, 977–982
  - tuple approach, 977–982
- Vertical fragmentation, DDBs, 844, 848–849
- Video source, multimedia databases, 996
- View definition language, 39
- View merging transformation, subqueries, 704–706
- Views
  - database designer development of, 15
  - equivalence, schedules, 771–772
  - serializability, schedules, 771–772
  - support of multiple data, 13
- Views (virtual tables)
  - authorization using, 232
  - base tables compared to, 228
  - CREATE VIEW statement, 228–229
  - data warehouses compared to, 1115
  - defining tables of, 228
  - hierarchical, 447–452
  - in-line, 232
  - DROP VIEW command, 229
  - materialization, 230
  - query modification for, 229–230
  - query retrieval using, 230–231
  - SQL virtual tables, 228–232
  - update strategies for, 230–232
  - virtual data in, 13
  - WITH CHECK option for, 232
  - XML document extraction and, 447–452
- Virtual data, 13
- Virtual private database (VPD)
  - technology, 1156
- Virtual relations (tables), 82
- Virtual storage access method (VSAM), 541
- Virtual tables, 228–232. *See also* Views (virtual tables)
- Visible attributes, objects, 371, 375
- Volatile/nonvolatile storage, 545
- Voldemort key-value data store, 897–899
- Weak entity types, 79, 292–293
- Web analytics, 1057
- Web-based user interfaces, 40
- Web crawlers, 1057
- Web database programming
  - HTML and, 343–346
  - Java technologies for, 358
  - PHP for, 343–359
- Web database systems
  - access control policies, 1141–1142
  - data interchanging using XML, 25
  - HTML and, 25
  - menu-based interfaces, 40
  - n*-tier architecture for, 49–51
  - security, 1141–1142
  - three-tier architecture for, 49–51
- Web information integration, 1052
- Web pages
  - hypertext documents for, 425
  - segmentation and noise reduction, 1053
  - XML and formatting of, 425–426
- Web search
  - defined, 1028
  - digital libraries for, 1047–1048
  - HITS ranking algorithm, 1051
  - link structure analysis, 1050–1051
  - PageRank ranking algorithm, 1051
  - search engines for, 1047
  - Web analysis and, 1048–1049
  - Web context analysis, 1051–1054
  - Web structure analysis, 1049–1050
  - Web usage analysis, 1054–1057
- Web servers
  - client/server architecture, 47
  - three-tier architecture, 50
- Web Services Description Language (WSDL), 447
- Web spamming, 1057
- Well-formed documents, XML, 433–424
- WHERE clause
  - asterisk (\*) for all attributes, 193
  - explicit set of values in, 214–215
  - grouping and, 221–222
  - SQL query retrieval and, 188–189, 192–193, 197, 214–215
  - selection (Boolean) condition of, 189
  - unspecified, 192–193
  - XQuery, 446
- WHERE CURRENT OF clause, SQL, 318
- Wide area network, 842
- Wildcard (\*) queries, 1036–1037
- WITH CHECK option, SQL views, 232
- WITH clause, SQL, 222–223
- Wrapper, 1025
- Write-ahead logging (WAL), database recovery, 816–818
- XML (EXtended Markup Language)
  - access control, 1140–1141
  - data models, 34, 51, 53
  - database extraction of documents, 442–443, 447–453
  - document type definition (DTD), 434–436
  - documents, 433–436, 442–443, 447–453
  - hierarchical (tree) data models, 51, 430–433, 447–453
  - hypertext documents and, 425
  - OpenPGP (Pretty Good Privacy) protocol, 1140–1141
  - protocols for, 446–447
  - query languages, 443–447
  - relational data model for document extraction, 447–449
  - schema language, 434, 436–441
  - semistructured data, 426–428
  - SQL functions for creation of data, 453–455
  - structured data, 426
  - tag notation and use, HTML, 428–430
  - unstructured data, 428–430
  - Web data interchanging using, 25
  - Web page formatting by, 425–426
  - XPath for path expressions, 443–445
  - XQuery, 445–446
- XPath, XML path expressions, 443–445
- XQuery, XML query specifications, 445–446
- YARN (Hadoop v2)
  - architecture, 940–942
  - Big data technology for, 936–944, 949–953
  - frameworks on, 943–944
  - rational behind development of, 937–939