APPENDIX

F File Organizations and Indexes

Objectives

In this appendix you will learn:

- The distinction between primary and secondary storage.
- The meanings of file organization and access method.
- How heap files are organized.
- How sequential files are organized.
- How hash files are organized.
- What an index is and how it can be used to speed up database retrievals.
- The distinction between a primary, secondary, and clustered indexes.
- How indexed sequential files are organized.
- How multilevel indexes are organized.
- How B^+-trees are organized.
- How bitmap indexes are organized.
- How join indexes are organized.
- How indexed clusters and hash clusters are organized.
- How to select an appropriate file organization.

Steps 4.2 and 4.3 of the physical database design methodology presented in Chapter 18 require the selection of appropriate file organizations and indexes for the base relations that have been created to represent the part of the enterprise being modeled. In this appendix we introduce the main concepts regarding the physical storage of the database on **secondary storage** devices such as magnetic disks and optical disks. The computer's **primary storage**—that is, main memory—is inappropriate for storing the database. Although the access times for primary storage are much faster than secondary storage, primary storage is not large or reliable enough to store the quantity of data that a typical database might require. Because the data stored in primary storage disappears when power is lost, we refer to primary storage as **volatile storage**. In contrast, the data on secondary storage persists through power loss, and is consequently referred to as **nonvolatile storage**.

In addition, the cost of storage per unit of data is an order of magnitude greater for primary storage than for disk storage.

Structure of this Appendix In Section F.1 we introduce the basic concepts of physical storage. In Sections F.2–F.4 we discuss the main types of file organization: heap (unordered), sequential (ordered), and hash files. In Section F.5 we discuss how indexes can be used to improve the performance of database retrievals. In particular, we examine indexed sequential files, multilevel indexes, B^+-trees, bitmap indexes, and join indexes. Finally, in Section F.6, we provide guidelines for selecting file organizations. The examples in this chapter are drawn from the *DreamHome* case study documented in Section 11.4 and Appendix A.

F.1 Basic Concepts

The database on secondary storage is organized into one or more **files**; each file consists of one or more **records** and each record consists of one or more **fields**. Typically, a record corresponds to an entity and a field to an attribute. Consider the reduced Staff relation from the *DreamHome* case study shown in Figure F.1.

We may expect each tuple in this relation to map to a record in the operating system file that holds the Staff relation. Each field in a record would store one attribute value from the Staff relation. When a user requests a tuple from the DBMS—for example, Staff tuple SG37—the DBMS maps this **logical record** on to a **physical record** and retrieves the physical record into the DBMS **buffers** in primary storage using the operating system file access routines.

The physical record is the unit of transfer between disk and primary storage, and vice versa. Generally, a physical record consists of more than one logical record, although depending on size, a logical record can correspond to one physical record. It is even possible for a large logical record to span more than one physical record. The terms **block** and **page** are sometimes used in place of physical record. In the remainder of this appendix we use the term "page." For example, the Staff tuples in Figure F.1 may be stored on two pages, as shown in Figure F.2.

staffNo	lName	position	branchNo
SL21	White	Manager	B005
SG37	Beech	Assistant	B003
SG14	Ford	Supervisor	B003
SA9	Howe	Assistant	B007
SG5	Brand	Manager	B003
SL41	Lee	Assistant	B005

Figure F.1 Reduced Staff relation from *DreamHome* case study.

staffNo	lName	position	branchNo	Page
SL21	White	Manager	B005	
SG37	Beech	Assistant	B003	1
SG14	Ford	Supervisor	B003	
SA9	Howe	Assistant	B007	
SG5	Brand	Manager	B003	2
SL41	Lee	Assistant	B005	

Figure F.2 Storage of Staff relation in pages.

The order in which records are stored and accessed in the file is dependent on the *file organization*.

File Organization The physical arrangement of data in a file into records and pages on secondary storage.

The main types of file organization are:

- **Heap** (**unordered**) files: Records are placed on disk in no particular order.
- **Sequential** (**ordered**) files: Records are ordered by the value of a specified field.
- **Hash** files: Records are placed on disk according to a hash function.

Along with a file organization, there is a set of *access methods*.

Access method The steps involved in storing and retrieving records from a file.

Because some access methods can be applied only to certain file organizations (for example, we cannot apply an indexed access method to a file without an index), the terms *file organization* and *access method* are used interchangeably. In the remainder of this appendix, we discuss the main types of file organization and access techniques and provide guidelines for their use.

F.2 Unordered Files

A **unordered file**, sometimes called a **heap file**, is the simplest type of file organization. Records are placed in the file in the same order as they are inserted. A new record is inserted in the last page of the file; if there is insufficient space in the last page, a new page is added to the file. This process makes insertion very efficient. However, as a heap file has no particular ordering with respect to field values, a **linear search** must be performed to access a record. A linear search involves reading pages from the file until the required record is found. This makes retrievals from heap files that have more than a few pages relatively slow, unless the retrieval involves a large proportion of the records in the file.

To delete a record, the required page first has to be retrieved, the record marked as deleted, and the page written back to disk. The space with deleted records is not reused. Consequently, performance progressively deteriorates as deletions occur. This means that heap files have to be periodically reorganized by the DBA to reclaim the unused space of deleted records.

Heap files are one of the best organizations for bulk loading data into a table. Because records are inserted at the end of the sequence, there is no overhead of calculating what page the record should go on.

F.3 Ordered Files

The records in a file can be sorted on the values of one or more of the fields, forming a key-sequenced data set. The resulting file is called an **ordered** or **sequential** file. The field(s) that the file is sorted on is called the **ordering field**. If the ordering

field is also a key of the file, and therefore guaranteed to have a unique value in each record, the field is also called the **ordering key** for the file. For example, consider the following SQL query:

```
SELECT *
FROM Staff
ORDER BY staffNo;
```

If the tuples of the Staff relation are already ordered according to the ordering field staffNo, it should be possible to reduce the execution time for the query, as no sorting is necessary. (Although in Section 4.2 we stated that tuples are unordered, this applies as an external (logical) property, not as an implementation or physical property. There will always be a first record, second record, and *n*th record.) If the tuples are ordered on staffNo, under certain conditions we can use a **binary search** to execute queries that involve a search condition based on staffNo. For example, consider the following SQL query:

```
SELECT *
FROM Staff
WHERE staffNo = 'SG37';
```

If we use the sample tuples shown in Figure F.1 and for simplicity assume there is one record per page, we would get the ordered file shown in Figure F.3. The binary search proceeds as follows:

(1) Retrieve the mid-page of the file. Check whether the required record is between the first and last records of this page. If so, the required record lies on this page and no more pages need to be retrieved.

(2) If the value of the key field in the first record on the page is greater than the required value, the required value, if it exists, occurs on an earlier page. Therefore, we repeat the previous steps using the lower half of the file as the new search area.

(3) If the value of the key field in the last record on the page is less than the required value, the required value occurs on a later page, and so we repeat the previous steps using the top half of the file as the new search area. In this way, half the search space is eliminated from the search with each page retrieved.

In our case, the middle page is page 3, and the record on the retrieved page (SG14) does not equal the one we want (SG37). The value of the key field in page 3 is less than the one we want, so we can discard the first half of the file from the search. We now retrieve the mid-page of the top half of the file, that is, page 5. This time the value of the key field (SL21) is greater than SG37, which enables us

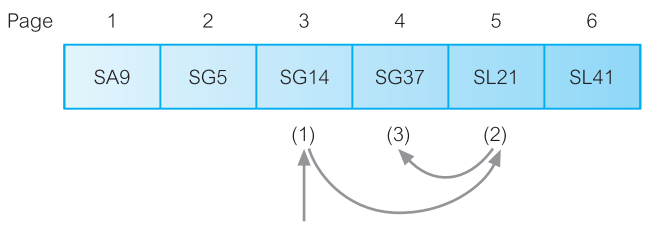


Figure F.3 Binary search on an ordered file.

to discard the top half of this search space. We now retrieve the mid-page of the remaining search space, that is, page 4, which is the record we want.

In general, the binary search is more efficient than a linear search. However, binary search is applied more frequently to data in primary storage than secondary storage.

Inserting and deleting records in a sorted file are problematic because the order of records has to be maintained. To insert a new record, we must find the correct position in the ordering for the record and then find space to insert it. If there is sufficient space in the required page for the new record, then the single page can be reordered and written back to disk. If this is not the case, then it would be necessary to move one or more records on to the next page. Again, the next page may have no free space and the records on this page must be moved, and so on.

Inserting a record near the start of a large file could be very time-consuming. One solution is to create a temporary unsorted file, called an *overflow* (or *transaction*) *file*. Insertions are added to the overflow file, and periodically, the overflow file is merged with the main sorted file. This makes insertions very efficient, but has a detrimental effect on retrievals. If the record is not found during the binary search, the overflow file has to be searched linearly. Inversely, to delete a record we must reorganize the records to remove the now free slot.

Ordered files are rarely used for database storage unless a primary index is added to the file (see Section F.5.1).

F.4 Hash Files

In a hash file, records do not have to be written sequentially to the file. Instead, a **hash function** calculates the address of the page in which the record is to be stored based on one or more fields in the record. The base field is called the **hash field**, or if the field is also a key field of the file, it is called the **hash key**. Records in a hash file will appear to be randomly distributed across the available file space. For this reason, hash files are sometimes called **random**, or **direct**, **files**.

The hash function is chosen so that records are as evenly distributed as possible throughout the file. One technique, called **folding**, applies an arithmetic function, such as addition, to different parts of the hash field. Character strings are converted into integers before the function is applied using some type of code, such as alphabetic position or ASCII values. For example, we could take the first two characters of the staff number, staffNo, convert them to an integer value, then add this value to the remaining digits of the field. The resulting sum is used as the address of the disk page in which the record is stored. An alternative, more popular technique, is the *division-remainder* hashing. This technique uses the MOD function, which takes the field value, divides it by some predetermined integer value, and uses the remainder of this division as the disk address.

The problem with most hashing functions is that they do not guarantee a unique address, because the number of possible values a hash field can take is typically much larger than the number of available addresses for records. Each address generated by a hashing function corresponds to a page, or **bucket**, with **slots** for multiple records. Within a bucket, records are placed in order of arrival. When the same address is generated for two or more records, a **collision** is said to have

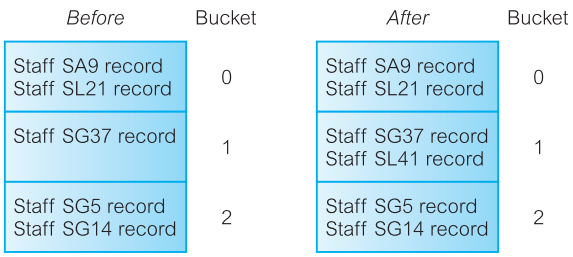


Figure F.4 Collision resolution using open addressing.

occurred (the records are called **synonyms**). In this situation, we must insert the new record in another position, because its hash address is occupied. Collision management complicates hash file management and degrades overall performance. There are several techniques that can be used to manage collisions:

- open addressing
- unchained overflow
- chained overflow
- multiple hashing

Open addressing

If a collision occurs, the system performs a linear search to find the first available slot to insert the new record. When the last bucket has been searched, the system starts back at the first bucket. Searching for a record employs the same technique used to store a record, except that the record is considered not to exist when an unused slot is encountered before the record has been located. For example, assume we have a trivial hash function that takes the digits of the staff number MOD 3, as shown in Figure F.4. Each bucket has two slots and staff records SG5 and SG14 hash to bucket 2. When record SL41 is inserted, the hash function generates an address corresponding to bucket 2. As there are no free slots in bucket 2, it searches for the first free slot, which it finds in bucket 1, after looping back and searching bucket 0.

Unchained overflow

Instead of searching for a free slot, an overflow area is maintained for collisions that cannot be placed at the hash address. Figure F.5 shows how the collision illustrated in Figure F.4 would be handled using an overflow area. In this case,

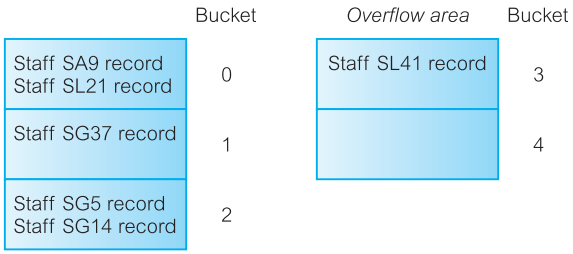


Figure F.5 Collision resolution using overflow.

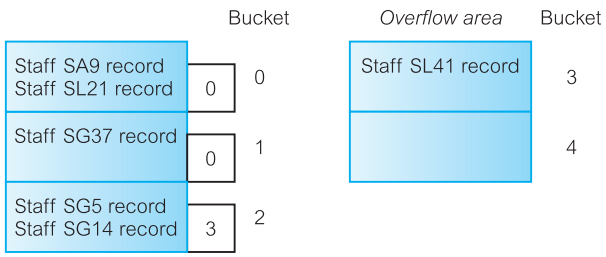


Figure F.6 Collision resolution using chained overflow.

instead of searching for a free slot for record SL41, the record is placed in the overflow area. At first sight, this may appear not to offer much performance improvement. However, using open addressing, collisions are located in the first free slot, potentially causing additional collisions in the future with records that hash to the address of the free slot. Thus, the number of collisions that occur is increased and performance is degraded. On the other hand, if we can minimize the number of collisions, it will be faster to perform a linear search on a smaller overflow area.

Chained overflow

As with the previous technique, an overflow area is maintained for collisions that cannot be placed at the hash address. However, with this technique each bucket has an additional field, sometimes called a **synonym pointer**, that indicates whether a collision has occurred and, if so, points to the overflow page used. If the pointer is zero, no collision has occurred. In Figure F.6, bucket 2 points to an overflow bucket 3; buckets 0 and 1 have a 0 pointer to indicate that there have been no collisions with these buckets yet.

A variation of this technique provides faster access to the overflow record by using a synonym pointer that points to a slot address within the overflow area rather than a bucket address. Records in the overflow area also have a synonym pointer that gives the address in the overflow area of the next synonym for the same target address, so that all synonyms for a particular address can be retrieved by following a chain of pointers.

Multiple hashing

An alternative approach to collision management is to apply a second hashing function if the first one results in a collision. The aim is to produce a new hash address that will avoid a collision. The second hashing function is generally used to place records in an overflow area.

With hashing, a record can be located efficiently by first applying the hash function and, if a collision has occurred, using one of these approaches to locate its new address. To update a hashed record, the record first has to be located. If the field to be updated is not the hash key, the update can take place and the record written back to the same slot. However, if the hash field is being updated, the hash function has to be applied to the new value. If a new hash address is generated, the record has to be deleted from its current slot and stored at its new address.

F.4.1 Dynamic Hashing

The previously described hashing techniques are *static*, in that the hash address space is fixed when the file is created. When the space becomes too full, it is said to be saturated, and the DBA must reorganize the hash structure. This may involve creating a new file with more space, choosing a new hashing function, and mapping the old file to the new file. An alternative approach is **dynamic hashing**, which allows the file size to change dynamically to accommodate growth and shrinkage of the database.

There have been many different dynamic hashing techniques proposed (see, for example, Larson, 1978; Fagin et al., 1979; Litwin, 1980). The basic principle of dynamic hashing is to manipulate the number generated by the hash function as a bit sequence, and to allocate records to buckets based on the progressive digitization of this sequence. A dynamic hash function generates values over a large range, namely b-bit binary integers, where b is typically 32. We briefly describe one type of dynamic hashing called **extendable hashing**.

Buckets are created as required. Initially, records are added to the first bucket until the bucket becomes full, at which point we split the bucket up depending on i bits of the hash value, where $0 \leq i < b$. These i bits are used as an offset into a **Bucket Address Table** (**BAT**), or directory. The value of i changes as the size of the database changes. The directory has a header that stores the current value of i, called the depth, together with 2^i pointers. Similarly, for each bucket there is a local depth indicator that specifies the value of i used to determine this bucket address. Figure F.7 shows an example of extendable hashing. We assume that each bucket has space for two records and the hash function uses the numerical part of the staff number, staffNo.

Figure F.7(a) shows the directory and bucket 0 after staff records SL21 and SG37 have been inserted. When we come to insert record SG14, bucket 0 is full, so we have to split bucket 0 based on the most significant bit of the hash value, as shown in Figure F.7(b). The directory contains 2^1 pointers for the bit values 0 and 1 ($i = 1$). The depth of the directory and the local depth of each bucket become 1. Again, when we come to insert the next record SA9, bucket 0 is again full, so we have to split the bucket based on the two most significant bits of the hash value, as shown in Figure F.7(c). The directory contains 2^2 pointers for the bit values 00, 01, 10, and 11 ($i = 2$). The depth of the directory and the local depth of buckets 0 and 2 become 2. Note that this does not affect bucket 1, so the directory for bits 10 and 11 both point to this bucket, and the local depth pointer for bucket 1 remains at 1.

When a bucket becomes empty after a deletion, it can be deleted together with its entry in the directory. In some schemes, it is possible to merge small buckets together and cut the size of the directory by half.

F.4.2 Limitations of Hashing

The use of hashing for retrievals depends upon the complete hash field. In general, hashing is inappropriate for retrievals based on pattern matching or ranges of values. For example, to search for values of the hash field in a specified range, we require a hash function that preserves order: that is, if r_{min} and r_{max} are minimum and maximum range values, then we require a hash function h, such that

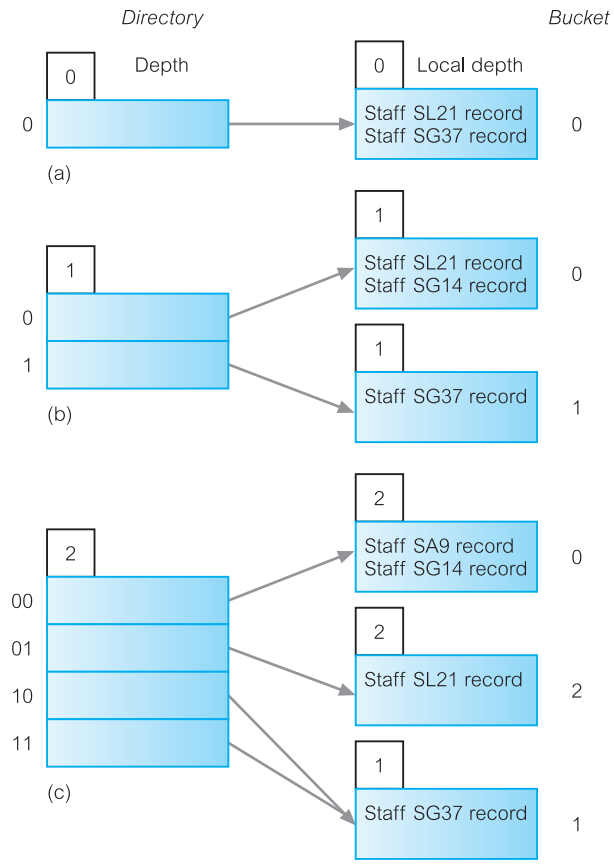


Figure F.7 Example of extendible hashing: (a) after insert of SL21 and SG37; (b) after insert of SG14; (c) after insert of SA9.

$h(r_{min}) < h(r_{max})$. In addition, hashing is inappropriate for retrievals based on a field other than the hash field. For example, if the Staff table is hashed on staffNo, then hashing could not be used to search for a record based on the IName field. In this case, it would be necessary to perform a linear search to find the record, or add IName as a secondary index (see Section F.5.3).

F.5 Indexes

In this section we discuss techniques for making the retrieval of data more efficient using **indexes**.

Index A data structure that allows the DBMS to locate particular records in a file more quickly and thereby speed response to user queries.

An index in a database is similar to an index in a book. It is an auxiliary structure associated with a file that can be referred to when searching for an item of information, just like searching the index of a book, in which we look up a keyword to get a list of one or more pages on which the keyword appears. An index obviates the need to scan sequentially through the file each time we want to find the item. In the case of database indexes, the required item will be one or more

records in a file. As in the book index analogy, the index is ordered, and each index entry contains the item required and one or more locations (record identifiers) where the item can be found.

Although indexes are not strictly necessary to use the DBMS, they can have a significant impact on performance. As with the book index, we could find the desired keyword by looking through the entire book, but this approach would be tedious and time-consuming. Having an index at the back of the book in alphabetical order of keyword allows us to go directly to the page or pages we want.

An index structure is associated with a particular search key and contains records consisting of the key value and the address of the logical record in the file containing the key value. The file containing the logical records is called the *data file* and the file containing the index records is called the *index file*. The values in the index file are ordered according to the *indexing field*, which is usually based on a single attribute.

F.5.1 Types of Index

There are different types of index, the main ones being:

- **Primary index.** The data file is sequentially ordered by an ordering key field (see Section F.3), and the indexing field is built on the ordering key field, which is guaranteed to have a unique value in each record.
- **Clustering index.** The data file is sequentially ordered on a non-key field, and the indexing field is built on this non-key field, so that there can be more than one record corresponding to a value of the indexing field. The non-key field is called a *clustering attribute*.
- **Secondary index.** An index that is defined on a non-ordering field of the data file.

A file can have *at most* one primary index or one clustering index, and in addition can have several secondary indexes. In addition, an index can be **sparse** or **dense**: a sparse index has an index record for only some of the search key values in the file; a dense index has an index record for every search key value in the file.

The search key for an index can consist of one or more fields. Figure F.8 illustrates four dense indexes on the (reduced) Staff table: one based on the salary column, one based on the branchNo column, one based on the composite index (salary, branchNo), and one based on the composite index (branchNo, salary).

F.5.2 Indexed Sequential Files

A sorted data file with a primary index is called an **indexed sequential file**. This structure is a compromise between a purely sequential file and a purely random file, in that records can be processed sequentially or individually accessed using a search key value that accesses the record via the index. An indexed sequential file is a more versatile structure, which normally has:

- a primary storage area
- a separate index or indexes
- an overflow area

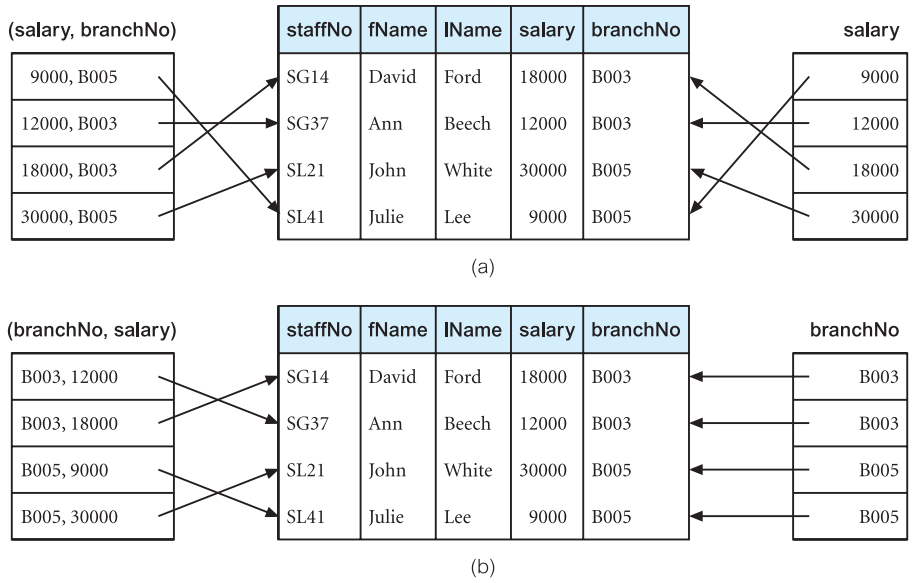


Figure F.8 Indexes on the Staff table: (a) (salary, branchNo) and salary; (b) (branchNo, salary) and branchNo.

IBM's Indexed Sequential Access Method (ISAM) uses this structure and is closely related to the underlying hardware characteristics. Periodically, these types of file need reorganizing to maintain efficiency. Reorganization is not only expensive but makes the file unavailable while it takes place. The later development, Virtual Sequential Access Method (VSAM), is an improvement on ISAM, in that it is hardware-independent. There is no separate designated overflow area, but there is space allocated in the data area to allow for expansion. As the file grows and shrinks, the process is handled dynamically without the need for periodic reorganization. Figure F.9(a) illustrates a dense index on a sorted file of Staff records. However, as the records in the data file are sorted, we can reduce the index to a sparse index as shown in Figure F.9(b).

Typically, a large part of a primary index can be stored in main memory and processed faster. Access methods, such as the binary search method discussed in Section F.3, can be used to further speed up the access. The main disadvantage of using a primary index, as with any sorted file, is maintaining the order as we insert and delete records. These problems are compounded as we have to maintain the

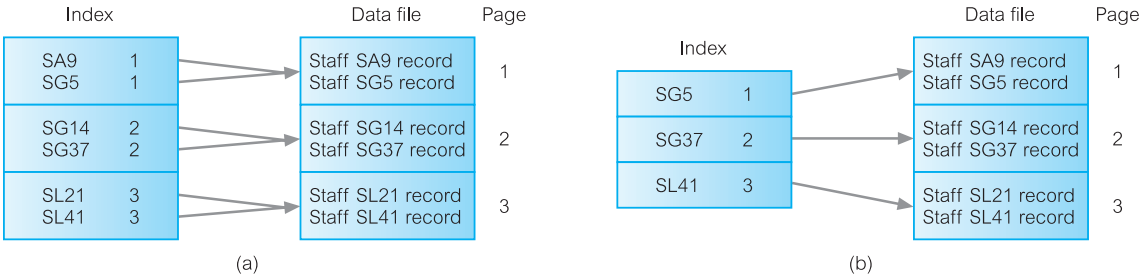


Figure F.9 Example of dense and sparse indexes: (a) dense index; (b) sparse index.

sorted order in the data file and in the index file. One method that can be used is the maintenance of an overflow area and chained pointers, similar to the technique described in Section F.4 for the management of collisions in hash files.

F.5.3 Secondary Indexes

A secondary index is also an ordered file similar to a primary index. However, whereas the data file associated with a primary index is sorted on the index key, the data file associated with a secondary index may not be sorted on the indexing key. Further, the secondary index key need not contain unique values, unlike a primary index. For example, we may wish to create a secondary index on the branchNo column of the Staff table but from Figure F.1 we can see that the values in the branchNo column are not unique. There are several techniques for handling nonunique secondary indexes:

- Produce a dense secondary index that maps on to all records in the data file, thereby allowing duplicate key values to appear in the index.
- Allow the secondary index to have an index entry for each distinct key value, but allow the block pointers to be multi-valued, with an entry corresponding to each duplicate key value in the data file.
- Allow the secondary index to have an index entry for each distinct key value. However, the block pointer would not point to the data file but to a bucket that contains pointers to the corresponding records in the data file.

Secondary indexes improve the performance of queries that use attributes other than the primary key. However, the improvement to queries has to be balanced against the overhead involved in maintaining the indexes while the database is being updated. This is part of physical database design and was discussed in Chapter 18.

F.5.4 Multilevel Indexes

When an index file becomes large and extends over many pages, the search time for the required index increases. For example, a binary search requires approximately $\log_2 p$ page accesses for an index with p pages. A **multilevel index** attempts to overcome this problem by reducing the search range. It does this by treating the index like any other file, splits the index into a number of smaller indexes, and maintains an index to the indexes. Figure F.10 shows an example of a two-level partial index for the Staff table of Figure F.1. Each page in the data file can store two records. For illustration, there are also two index records per page, although in practice there would be many index records per page. Each index record stores

Figure F.10 Example of a multilevel index.

an access key value and a page address. The stored access key value is the highest in the addressed page.

To locate a record with a specified staffNo value, say SG14, we start from the second-level index and search the page for the last access key value that is less than or equal to SG14, in this case SG37. This record contains an address to the first-level index page to continue the search. Repeating the process leads to page 2 in the data file, where the record is stored. If a range of staffNo values had been specified, we could use the same process to locate the first record in the data file corresponding to the lower range value. As the records in the data file are sorted on staffNo, we can find the remaining records in the range by reading serially through the data file.

IBM's ISAM is based on a two-level index structure. Insertion is handled by overflow pages, as discussed in Section F.4. In general, an n-level index can be built, although three levels are common in practice; a file would have to be very large to require more than three levels. In the following section we discuss a particular type of multilevel dense index called a **B$^+$-tree**.

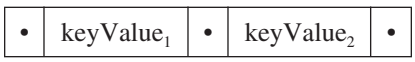
F.5.5 B$^+$-trees

Many DBMSs use a data structure called a **tree** to hold data or indexes. A tree consists of a hierarchy of nodes. Each node in the tree, except the **root** node, has one **parent** node and zero or more **child** nodes. A root node has no parent. A node that does not have any children is called a **leaf** node.

The **depth** of a tree is the maximum number of levels between the root node and a leaf node in the tree. Depth may vary across different paths from root to leaf, or depth may be the same from the root node to each leaf node, producing a tree called a **balanced tree**, or **B-tree** (Bayer and McCreight, 1972; Comer, 1979). The **degree**, or **order**, of a tree is the maximum number of children allowed per parent. Large degrees, in general, create broader, shallower trees. Because access time in a tree structure depends more often upon depth than on breadth, it is usually advantageous to have "bushy," shallow trees. A binary tree has order 2 in which each node has no more than two children. The rules for a B$^+$-tree are as follows:

- If the root is not a leaf node, it must have at least two children.
- For a tree of order n, each node (except the root and leaf nodes) must have between $n/2$ and n pointers and children. If $n/2$ is not an integer, the result is rounded up.
- For a tree of order n, the number of key values in a leaf node must be between $(n - 1)/2$ and $(n - 1)$ pointers and children. If $(n - 1)/2$ is not an integer, the result is rounded up.
- The number of key values contained in a nonleaf node is 1 less than the number of pointers.
- The tree must always be balanced: that is, every path from the root node to a leaf must have the same length.
- Leaf nodes are linked in order of key values.

Figure F.11 represents an index on the staffNo field of the Staff table in Figure F.1 as a B$^+$-tree of order 1. Each node is of the form:



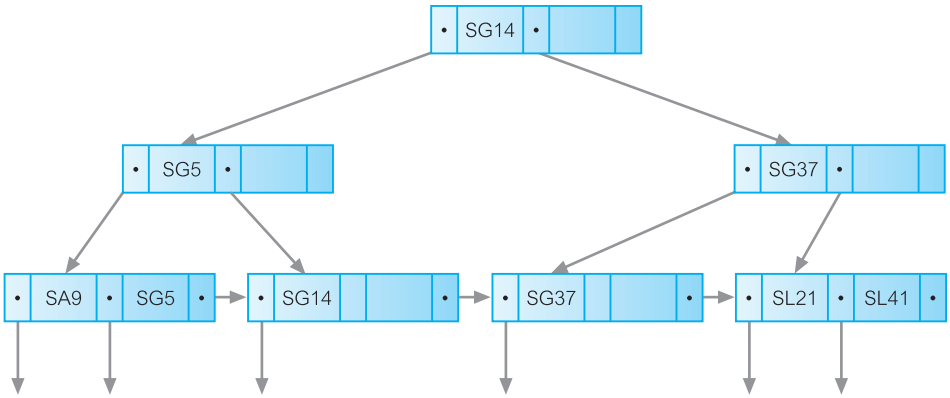


Figure F.11 Example of B+-tree index.

where • can be blank or represent a pointer to another record. If the search key value is less than or equal to key $Value_i$, the pointer to the left of key $Value_i$ is used to find the next node to be searched; otherwise, the pointer at the end of the node is used. For example, to locate SL21, we start from the root node. SL21 is greater than SG14, so we follow the pointer to the right, which leads to the second-level node containing the key values SG37 and SL21. We follow the pointer to the left of SL21, which leads to the leaf node containing the address of record SL21.

In practice, each node in the tree is actually a page, so we can store more than three pointers and two key values. If we assume that a page has 4096 bytes, each pointer is 4 bytes long and the staffNo field requires 4 bytes of storage, and each page has a 4-byte pointer to the next node on the same level, we could store (4096 − 4)/(4 + 4) = 511 index records per page. The B+-tree would be order 512. The root can store 511 records and can have 512 children. Each child can also store 511 records, giving a total of 261,632 records. Each child can also have 512 children, giving a total of 262,144 children on level 2 of the tree. Each of these children can have 511 records giving a total of 133,955,584. This gives a theoretical maximum number of index records as:

root:	511
Level 1:	261,632
Level 2:	133,955,584
TOTAL	134,217,727

Thus, we could randomly access one record in the Staff file containing 134,217,727 records within four disk accesses (in fact, the root would normally be stored in main memory, so there would be one fewer disk access). In practice, however, the number of records held in each page would be smaller, as not all pages would be full (see Figure F.11).

A B+-tree always takes approximately the same time to access any data record, by ensuring that the same number of nodes is searched: in other words, by ensuring that the tree has a constant depth. Being a dense index, every record is addressed by the index so there is no requirement for the data file to be sorted; for example, it could

be stored as a heap file. However, balancing can be costly to maintain as the tree contents are updated. Figure F.12 provides a worked example of how a B+-tree would be maintained as records are inserted using the order of the records in Figure F.1.

Figure F.12(a) shows the construction of the tree after the insertion of the first two records SL21 and SG37. The next record to be inserted is SG14. The node is full, so we must split the node by moving SL21 to a new node. In addition, we create a parent node consisting of the rightmost key value of the left node, as shown in Figure F.12(b). The next record to be inserted is SA9. SA9 should be located to the left of SG14, but again the node is full. We split the node by moving SG37 to a new node. We also move SG14 to the parent node, as shown in Figure F.12(c). The next record to be inserted is SG5. SG5 should be located to the right of SA9,

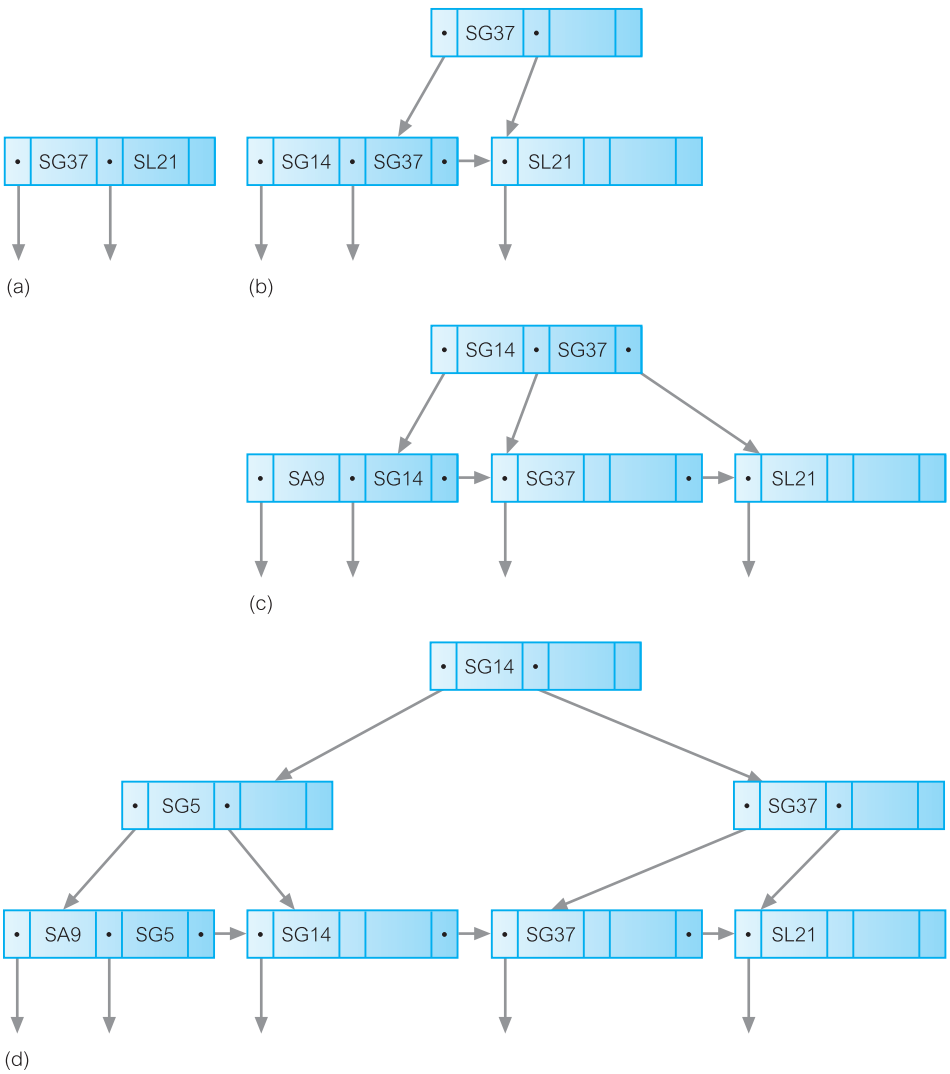


Figure F.12 Insertions into a B+-tree index: (a) after insertion of SL21, SG37; (b) after insertion of SG14; (c) after insertion of SA9; (d) after insertion of SG5.

but again the node is full. We split the node by moving SG14 to a new node and add SG5 to the parent node. However, the parent node is also full and has to be split. In addition, a new parent node has to be created, as shown in Figure F.12(d). Finally, record SL41 is added to the right of SL21, as shown in Figure F.11.

F.5.6 Bitmap Indexes

Another type of index that is becoming increasingly popular, particularly in data warehousing, is the **bitmap index**. Bitmap indexes are generally used on attributes that have a sparse domain (that is, the domain contains a relatively low number of possible values). Rather than storing the actual value of the attribute, the bitmap index stores a *bit vector* for each attribute indicating which tuples contain this particular domain value. Each bit that is set to 1 in the bitmap corresponds to a row identifier. If the number of different domain values is small, then bitmap indexes are very space-efficient.

For example, consider the Staff relation shown in Figure F.13(a). Assume that the position attribute can take only one of the values present (that is, Manager, Assistant, or Supervisor) and similarly assume that the branchNo attribute can take only one of the values present (that is, B003, B005, or B007). We could construct bitmap indexes to represent these two attributes as shown in Figure F.13(b).

Bitmap indexes provide two important advantages over B^+-tree indexes. First, they can be more compact than B^+-tree indexes, requiring less storage space, and they lend themselves to compression techniques. Second, bitmap indexes can provide significant performance improvements when the query involves multiple predicates each with its own bitmap index. For example, consider the following query:

```
SELECT staffNo, salary
FROM Staff
WHERE position = 'Supervisor' AND branchNo = 'B003';
```

Figure F.13
(a) Staff relation; (b) bitmap indexes on the position and branchNo attributes.

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

(a)

Manager	Assistant	Supervisor
1	0	0
0	1	0
0	0	1
0	1	0
1	0	0
0	1	0

B003	B005	B007
0	1	0
1	0	0
1	0	0
0	0	1
1	0	0
0	1	0

(b)

In this case, we can take the third bit vector for position and perform a bitwise AND with the first bit vector for branchNo to obtain a bit vector that has a 1 for every Supervisor who works at branch 'B003'.

F.5.7 Join Indexes

Another type of index that is becoming increasingly popular, also in data warehousing in particular, is the **join index**. A join index is an index on attributes from two or more relations that come from the same domain. For example, consider the extended Branch and PropertyForRent relations shown in Figure F.14(a). We could create a join index on the nonkey city attribute to generate the index relation shown in Figure F.14(b). We have chosen to sort the join index on the branchRowID,

Branch

rowID	branchNo	street	city	postcode
20001	B005	22 Deer Rd	London	SW1 4EH
20002	B007	16 Argyll St	Aberdeen	AB2 3SU
20003	B003	163 Main St	Glasgow	G11 9QX
20004	B004	32 Manse Rd	Bristol	BS99 1NZ
20005	B002	56 Clover Dr	London	NW10 6EU
20006	...			

PropertyForRent

rowID	propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
30001	PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
30002	PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
30003	PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
30004	PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
30005	PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
30006	PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003
30007	...									

(a)

Join Index

branchRowID	propertyRowID	city
20001	30002	London
20002	30001	Aberdeen
20003	30003	Glasgow
20003	30004	Glasgow
20003	30005	Glasgow
20003	30006	Glasgow
20005	30002	London
20006	...	

(b)

Figure F.14 (a) Branch and PropertyForRent relations; (b) join index on the nonkey city attribute.

but it could have been sorted on any of the three attributes. Sometimes two join indexes are created, one as shown and one with the two rowID attributes reversed.

This type of query could be common in data warehousing applications when attempting to find out facts about related pieces of data (in this case, we are attempting to find how many properties come from a city that has an existing branch). The join index precomputes the join of the Branch and PropertyForRent relations based on the city attribute, thereby removing the need to perform the join each time the query is run, and improving the performance of the query. This could be particularly important if the query has a high frequency. Oracle combines the bitmap index and the join index to provide a **bitmap join index**.

F.6 Clustered and Nonclustered Tables

Some DBMSs, such as Oracle, support **clustered** and **nonclustered** tables. The choice of whether to use a clustered or nonclustered table depends on the analysis of the transactions undertaken previously, but the choice can have an impact on performance. In this section we briefly examine both types of structure.

Clusters are groups of one or more tables physically stored together because they share common columns and are often used together. With related records being physically stored together, disk access time is improved. The related columns of the tables in a cluster are called the **cluster key**. The cluster key is stored only once, so clusters store a set of tables more efficiently than if the tables were stored individually (not clustered).

Figure F.15 illustrates how the Branch and Staff tables would be stored if we clustered the tables based on the column branchNo. When these two tables are clustered, each unique branchNo value is stored only once, in the cluster key. To each branchNo value are attached the columns from both these tables.

As we now discuss, Oracle supports two types of clusters: indexed clusters and hash clusters.

F.6.1 Indexed Clusters

In an indexed cluster, records with the same cluster key are stored together. Oracle suggests using indexed clusters when:

street	city	postcode	branchNo	staffNo	fName	lName	position	sex	DOB	salary
22 Deer Rd	London	SW1 4EH	B005	SL21	John	White	Manager	…	…	30000
				SL41	Julie	Lee	Assistant			9000
163 Main St	Glasgow	G11 9QX	B003	SG37	Ann	Beech	Assistant	…	…	12000
				SG14	David	Ford	Supervisor			18000
				SG5	Susan	Brand	Manager			24000

Staff table

Branch table

Cluster key

Figure F.15 How the Branch and Staff tables would be stored clustered on branchNo.

- queries retrieve records over a range of cluster key values;
- clustered tables may grow unpredictably.

Clusters can improve performance of data retrieval, depending on the data distribution and what SQL operations are most often performed on the data. In particular, tables that are joined in a query benefit from the use of clusters, because the records common to the joined tables are retrieved with the same I/O operation.

To create an indexed cluster in Oracle called BranchIndexedCluster with the cluster key column branchNo, we could use the following SQL statement:

```
CREATE CLUSTER BranchIndexedCluster
  (branchNo CHAR(4))
SIZE 512
STORAGE (INITIAL 100K NEXT 50K PCTINCREASE 10);
```

The SIZE parameter specifies the amount of space (in bytes) to store all records with the same cluster key value. The size is optional and, if omitted, Oracle reserves one data block for each cluster key value. The INITIAL parameter specifies the size (in bytes) of the cluster's first extent, and the NEXT parameter specifies the size (in bytes) of the next extent to be allocated. The PCTINCREASE parameter specifies the percentage by which the third and subsequent extents grow over the preceding extent (default 50). In our example, we have specified that each subsequent extent should be 10% larger than the preceding extent.

F.6.2 Hash Clusters

Hash clusters also cluster table data in a manner similar to index clusters. However, a record is stored in a hash cluster based on the result of applying a hash function to the record's cluster key value. All records with the same hash key value are stored together on disk. Oracle suggests using hash clusters when:

- queries retrieve records based on equality conditions involving all cluster key columns (for example, return all records for branch B005);
- clustered tables are static or we can determine the maximum number of records and the maximum amount of space required by the cluster when it is created.

To create a hash cluster in Oracle called PropertyHashCluster clustered by the column propertyNo, we could use the following SQL statement:

```
CREATE CLUSTER PropertyHashCluster
  (propertyNo VARCHAR2(5))
  HASH IS propertyNo HASHKEYS 300000;
```

Once the hash cluster has been created, we can create the tables that will be part of the structure. For example:

```
CREATE TABLE PropertyForRent
  (propertyNo VARCHAR2(5) PRIMARY KEY,
…)
  CLUSTER PropertyHashCluster (propertyNo);
```

F.7 Guidelines for Selecting File Organizations

As an aid to understanding file organizations and indexes more fully, we provide guidelines for selecting a file organization based on the following types of file:

- Heap
- Hash
- Indexed Sequential Access Method (ISAM)
- B^+-tree
- Clusters

Heap (unordered)

The heap file organization is discussed in Appendix F.2. Heap is a good storage structure in the following situations:

(1) When data is being bulk-loaded into the relation. For example, to populate a relation after it has been created, a batch of tuples may have to be inserted into the relation. If heap is chosen as the initial file organization, it may be more efficient to restructure the file after the insertions have been completed.

(2) The relation is only a few pages long. In this case, the time to locate any tuple is short, even if the entire relation has to be searched serially.

(3) When every tuple in the relation has to be retrieved (in any order) every time the relation is accessed. For example, retrieve the addresses of all properties for rent.

(4) When the relation has an additional access structure, such as an index key, heap storage can be used to conserve space.

Heap files are inappropriate when only selected tuples of a relation are to be accessed.

Hash

The hash file organization is discussed in Appendix F.4. Hash is a good storage structure when tuples are retrieved based on an exact match on the hash field value, particularly if the access order is random. For example, if the PropertyForRent relation is hashed on propertyNo, retrieval of the tuple with propertyNo equal to PG36 is efficient. However, hash is not a good storage structure in the following situations:

(1) When tuples are retrieved based on a pattern match of the hash field value. For example, retrieve all properties whose property number, propertyNo, begins with the characters "PG."

(2) When tuples are retrieved based on a range of values for the hash field. For example, retrieve all properties with a rent in the range 300–500.

(3) When tuples are retrieved based on a field other than the hash field. For example, if the Staff relation is hashed on staffNo, then hashing cannot be used to search for a tuple based on the lName attribute. In this case, it would be necessary to perform a linear search to find the tuple, or add lName as a secondary index (see Step 4.3).

(4) When tuples are retrieved based on only part of the hash field. For example, if the PropertyForRent relation is hashed on rooms and rent, then hashing cannot

be used to search for a tuple based on the rooms attribute alone. Again, it would be necessary to perform a linear search to find the tuple.

(5) When the hash field is frequently updated. When a hash field is updated, the DBMS must delete the entire tuple and possibly relocate it to a new address (if the hash function results in a new address). Thus, frequent updating of the hash field affects performance.

Indexed Sequential Access Method (ISAM)

The indexed sequential file organization is discussed in Appendix F.5.2. ISAM is a more versatile storage structure than hash; it supports retrievals based on exact key match, pattern matching, range of values, and part key specification. However, the ISAM index is static, created when the file is created. Thus, the performance of an ISAM file deteriorates as the relation is updated. Updates also cause an ISAM file to lose the access key sequence, so that retrievals in order of the access key will become slower. These two problems are overcome by the B^+-tree file organization. However, unlike B^+-tree, concurrent access to the index can be easily managed, because the index is static.

B^+-tree

The B^+-tree file organization is discussed in Appendix F.5.5. Again, B^+-tree is a more versatile storage structure than hashing. It supports retrievals based on exact key match, pattern matching, range of values, and part key specification. The B^+-tree index is dynamic, growing as the relation grows. Thus, unlike ISAM, the performance of a B^+-tree file does not deteriorate as the relation is updated. The B^+-tree also maintains the order of the access key even when the file is updated, so retrieval of tuples in the order of the access key is more efficient than ISAM. However, if the relation is not frequently updated, the ISAM structure may be more efficient, as it has one fewer level of index than the B^+-tree, whose leaf nodes contain pointers to the actual tuples rather than the tuples themselves.

Clustered tables

Some DBMSs, for example Oracle, support **clustered tables** (see Appendix F.6). The choice of whether to use a clustered or nonclustered table depends on the analysis of the transactions undertaken previously, but the choice can have an impact on performance. Following, we provide guidelines for the use of clustered tables. Note in this section, we use the Oracle terminology, which refers to a relation as a *table* with *columns* and *rows*.

Clusters are groups of one or more tables physically stored together because they share common columns and are often used together. With related rows being physically stored together, disk access time is improved. The related columns of the tables in a cluster are called the **cluster key**. The cluster key is stored only once, so clusters store a set of tables more efficiently than if the tables were stored individually (not clustered). Oracle supports two types of clusters: indexed clusters and hash clusters.

(a) Indexed clusters In an indexed cluster, rows with the same cluster key are stored together. Oracle suggests using indexed clusters when:

- queries retrieve rows over a range of cluster key values;
- clustered tables may grow unpredictably.

The following guidelines may be helpful when deciding whether to cluster tables:

- Consider clustering tables that are often accessed in join statements.
- Do not cluster tables if they are joined only occasionally or their common column values are modified frequently. (Modifying a row's cluster key value takes longer than modifying the value in an unclustered table, because Oracle may have to migrate the modified row to another block to maintain the cluster.)
- Do not cluster tables if a full search of one of the tables is often required. (A full search of a clustered table can take longer than a full search of an unclustered table. Oracle is likely to read more blocks, because the tables are stored together.)
- Consider clustering tables involved in a one-to-many (1:*) relationship if a row is often selected from the parent table and then the corresponding rows from the child table. (Child rows are stored in the same data block(s) as the parent row, so they are likely to be in memory when selected, requiring Oracle to perform less I/O.)
- Consider storing a child table alone in a cluster if many child rows are selected from the same parent. (This measure improves the performance of queries that select child rows of the same parent but does not decrease the performance of a full search of the parent table.)
- Do not cluster tables if the data from all tables with the same cluster key value exceeds more than one or two Oracle blocks. (To access a row in a clustered table, Oracle reads all blocks containing rows with that value. If these rows occupy multiple blocks, accessing a single row could require more reads than accessing the same row in an unclustered table.)

(b) Hash clusters **Hash clusters** also cluster table data in a manner similar to index clusters. However, a row is stored in a hash cluster based on the result of applying a hash function to the row's cluster key value. All rows with the same hash key value are stored together on disk. Oracle suggests using hash clusters when:

- queries retrieve rows based on equality conditions involving all cluster key columns (for example, return all rows for branch B003);
- clustered tables are static or the maximum number of rows and the maximum amount of space required by the cluster can be determined when it is created.

The following guidelines may be helpful when deciding whether to use hash clusters:

- Consider using hash clusters to store tables that are frequently accessed using a search clause containing equality conditions with the same column(s). Designate these column(s) as the cluster key.
- Store a table in a hash cluster if it can be determined how much space is required to hold all rows with a given cluster key value, both now and in the future.
- Do not use hash clusters if space is scarce and it is not affordable to allocate additional space for rows to be inserted in the future.

- Do not use a hash cluster to store a constantly growing table if the process of occasionally creating a new, larger hash cluster to hold that table is impractical.
- Do not store a table in a hash cluster if a search of the entire table is often required and a significant amount of space must be allocated to the hash cluster in anticipation of the table growing. (Such full searches must read all blocks allocated to the hash cluster, even though some blocks may contain few rows. Storing the table alone would reduce the number of blocks read by a full table search.)
- Do not store a table in a hash cluster if the cluster key values are frequently modified.
- Storing a single table in a hash cluster can be useful, regardless of whether the table is often joined with other tables, provided that hashing is appropriate for the table based on the previous guidelines.

Appendix Summary

- A **file organization** is the physical arrangement of data in a file into records and pages of secondary storage. An **access method** is the steps involved in storing and retrieving records from a file.
- **Heap** (**unordered**) files store records in the same order they are inserted. Heap files are good for inserting a large number of records into the file. They are inappropriate when only selected records are to be retrieved.
- **Sequential** (**ordered**) files store records sorted on the values of one or more fields (the ordering fields). Inserting and deleting records in a sorted file is problematic, because the order of records has to be maintained. As a result, ordered files are rarely used for database storage unless a primary index is added to the file.
- **Hash** files are good when retrieval is based on an exact key match. They are not good when retrieval is based on pattern matching, range of values, part keys, or a column other than the hash field.
- An **index** is a data structure that allows the DBMS to locate particular records in a file more quickly and thereby speed response to user queries. There are three main types of index: a **primary index**, **clustering index**, and a **secondary index** (an index that is defined on a non-ordering field of the data file).
- **Secondary indexes** provide a mechanism for specifying an additional key for a base relation that can be used to retrieve data more efficiently. However, there is an overhead involved in the maintenance and use of secondary indexes that has to be balanced against the performance improvement gained when retrieving data.
- **ISAM** is more versatile than hashing, supporting retrievals based on exact key match, pattern matching, range of values, and part key specification. However, the ISAM index is static, so performance deteriorates as the table is updated. Updates also cause the ISAM file to lose the access key sequence, so retrievals in order of the access key become slower.
- These two problems are overcome by the **B$^+$-tree** file organization, which has a dynamic index. However, unlike B$^+$-tree, because the ISAM index is static, concurrent access to the index can be easily managed. If the relation is not frequently updated or not very large or likely to be, the ISAM structure may be more efficient as it has one less level of index than the B$^+$-tree, whose leaf nodes contain record pointers.
- A **bitmap index** stores a *bit vector* for each attribute indicating which tuples contain this particular domain value. Each bit that is set to 1 in the bitmap corresponds to a row identifier. If the number of different domain values is small, then bitmap indexes are very space efficient.

- A **join index** is an index on attributes from two or more relations that come from the same domain. The join index precomputes the join of the two relations based on the specified attribute, thereby removing the need to perform the join each time the query is run, and improving the performance of the query. This could be particularly important if the query has a high frequency.
- **Clusters** are groups of one or more tables physically stored together because they share common columns and are often used together. With related records being physically stored together, disk access time is improved. The related columns of the tables in a cluster are called the **cluster key**. The cluster key is stored only once, and so clusters store a set of tables more efficiently than if the tables were stored individually (not clustered). Oracle supports two types of clusters: indexed clusters and hash clusters.