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 18.3. 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.[11]

Variations of B-Trees and B$^+$-Trees. To conclude this section, we briefly mention some variations of B-trees and B$^+$-trees. In some cases, constraint 5 on the B-tree (or for the internal nodes of the B$^+$–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$^+$-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 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.

18.4 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.

[11]For more details on insertion and deletion algorithms for B$^+$ trees, consult Ramakrishnan and Gehrke [2003].

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. 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**.

18.4.1 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 $<A_1, A_2, \ldots, A_n>$, the search key values are tuples with n values: $<v_1, v_2, \ldots, v_n>$.

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 <3, n> precedes <4, m> for any values of m and n. The ascending key order for keys with Dno = 4 would be <4, 18>, <4, 19>, <4, 20>, 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.

18.4.2 Partitioned Hashing

Partitioned hashing is an extension of static external hashing (Section 17.8.2) 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 <Dno, Age>. If Dno and Age are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that Dno = 4 has a hash address '100' and Age = 59 has hash address '10101'. Then to search for the combined search value, Dno = 4 and Age = 59, one goes to bucket address 100 10101; just to search for all employees with Age = 59, all buckets (eight of them) will be searched whose addresses are '000 10101', '001 10101', ... and

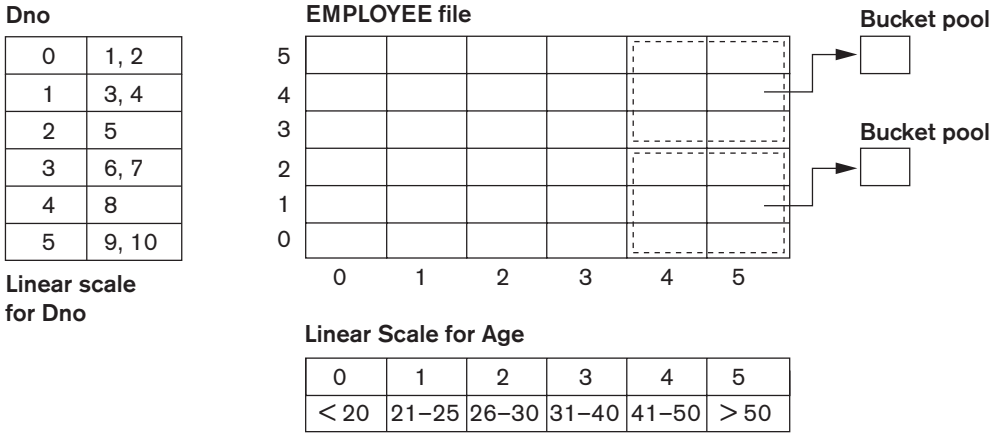
so on. An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high-order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

18.4.3 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 Dno and Age as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes. Figure 18.14 shows a grid array for the EMPLOYEE file with one linear scale for Dno and another for the 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 Dno has Dno = 1, 2 combined as one value 0 on the scale, while Dno = 5 corresponds to the value 2 on that scale. Similarly, 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 18.14 also shows the assignment of cells to buckets (only partially).

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 the 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

Figure 18.14
Example of a grid array on Dno and Age attributes.



and Age > 40 refers to the data in the top bucket shown in Figure 18.14. 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.[12]

18.5 Other Types of Indexes

18.5.1 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 <*K*, *Pr*> or <*K*, *P*>, 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 17.8.3; searching for an entry uses the hash search algorithm on *K*. Once an entry is found, the pointer *Pr* (or *P*) is used to locate the corresponding record in the data file. Figure 18.15 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 about dynamic hashing in Section 17.8.3. Other search structures can also be used as indexes.

18.5.2 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 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.

[12]Insertion/deletion algorithms for grid files may be found in Nievergelt et al. (1984).

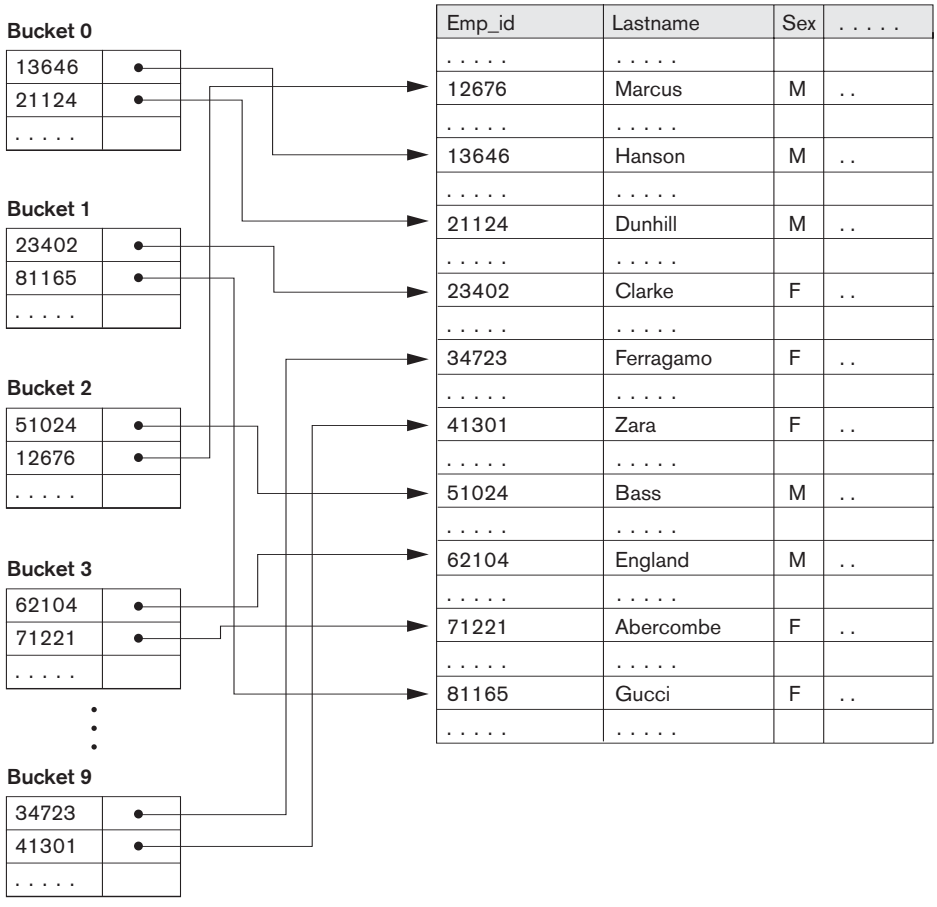


Figure 18.15
Hash-based indexing.

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. 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 $<v_1, v_2, ..., v_m>$ with m distinct values, then m bitmap indexes would be created for that column. Figure 18.16 shows the relation EMPLOYEE with columns Emp_id, Lname, Sex, Zipcode, and Salary_grade (with just 8 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.

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 18.16
Bitmap indexes for Sex and Zipcode

- 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.
- 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 in Figure 18.16. 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 percent of the records would actually have to be retrieved. If a column has only a few values, like the Sex column in Figure 18.16, retrieval of the Sex = M condition on average would retrieve 50 percent 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 percent 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 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. This process represents 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^+-Tree Leaf Nodes. Bitmaps can be used on the leaf nodes of B^+-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^+-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 percent 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.

18.5.3 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.[13]

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));
```

[13]Rafi Ahmed contributed most of this section.

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));
```

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.

18.6 Some General Issues Concerning Indexing

18.6.1 Logical versus Physical Indexes

In the earlier discussion, we have assumed that the index entries <*K*, *Pr*> (or <*K*, *P*>) 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 <K, K_p>. 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. 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.

18.6.2 Discussion

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. 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, 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.

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^+-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^+-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.

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^+-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 18.2. Another IBM method, the **virtual storage access method** (**VSAM**), is somewhat similar to the B^+–tree access structure and is still being used in many commercial systems.

18.6.3 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^+-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
- Use of column-wise indexes (similar to the bitmap indexes discussed in Section 18.5.2) and join indexes on multiple tables to answer queries without having to access the data tables

- Use of materialized views (see Chapter 5) 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. Further discussion on column-store DBMSs can be found in the references mentioned in this chapter's Selected Bibliography.

18.7 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 17, 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.

Next we showed how multilevel indexes can be implemented as B-trees and B^+-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 percent full, allowing space for insertions without requiring reorganization of the index for the majority of insertions. B^+-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 showed how an index can be constructed based on hash data structures. 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^+ 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 also reviewed some general issues related to indexing, and commented on column-based storage of relations, which has particular advantages for read-only databases. Finally, we discussed how combinations of the above organizations can be used. For example, secondary indexes are often used with mixed files, as well as with unordered and ordered files.

Review Questions

18.1. Define the following terms: *indexing field*, *primary key field*, *clustering field*, *secondary key field*, *block anchor*, *dense index*, and *nondense (sparse) index*.

18.2. 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?

18.3. Why can we have at most one primary or clustering index on a file, but several secondary indexes?

18.4. How does multilevel indexing improve the efficiency of searching an index file?

18.5. What is the order p of a B-tree? Describe the structure of B-tree nodes.

18.6. What is the order p of a B^+-tree? Describe the structure of both internal and leaf nodes of a B^+-tree.

18.7. How does a B-tree differ from a B^+-tree? Why is a B^+-tree usually preferred as an access structure to a data file?

18.8. Explain what alternative choices exist for accessing a file based on multiple search keys.

18.9. What is partitioned hashing? How does it work? What are its limitations?

18.10. What is a grid file? What are its advantages and disadvantages?

18.11. Show an example of constructing a grid array on two attributes on some file.

18.12. What is a fully inverted file? What is an indexed sequential file?

18.13. How can hashing be used to construct an index?

18.14. 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.

18.15. What is the concept of function-based indexing? What additional purpose does it serve?

18.16. What is the difference between a logical index and a physical index?

18.17. What is column-based storage of a relational database?

Exercises

18.18. 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 18.1.3, 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 percent full (rounded up for convenience); (iii) the number of levels needed if internal nodes are also 69 percent 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.

18.19. 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.

18.20. Repeat Exercise 18.19, but use a B-tree of order $p = 4$ instead of a B^+-tree.

18.21. Suppose that the following search field values are deleted, in the given order, from the B^+-tree of Exercise 18.19; show how the tree will shrink and show the final tree. The deleted values are 65, 75, 43, 18, 20, 92, 59, 37.

18.22. Repeat Exercise 18.21, but for the B-tree of Exercise 18.20.

18.23. Algorithm 18.1 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. A multilevel secondary index on a nonkey nonordering field of a file. Assume that option 3 of Section 18.1.3 is used, where an extra level of indirection stores pointers to the individual records with the corresponding index field value.

b. A multilevel secondary index on a nonordering key field of a file.

c. A multilevel clustering index on a nonkey ordering field of a file.

18.24. Suppose that several secondary indexes exist on nonkey fields of a file, implemented using option 3 of Section 18.1.3; for example, we could have secondary indexes on the fields Department_code, Job_code, and Salary of the EMPLOYEE file of Exercise 18.18. Describe an efficient way to search for and retrieve records satisfying a complex selection condition on these fields, such as (Department_code = 5 AND Job_code = 12 AND Salary = 50,000), using the record pointers in the indirection level.

18.25. Adapt Algorithms 18.2 and 18.3, which outline search and insertion procedures for a B^+-tree, to a B-tree.

18.26. 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 18.17 (next page) illustrates how this could be done for our example in Figure 18.12; 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 18.17 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.

18.27. Outline an algorithm for deletion from a B^+-tree.

18.28. Repeat Exercise 18.27 for a B-tree.

Selected Bibliography

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 Ramakrihnan and Gehrke (2003) discuss indexing in detail and may be consulted for search, insertion, and deletion algorithms for B-trees and B^+-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^+-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^+-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.

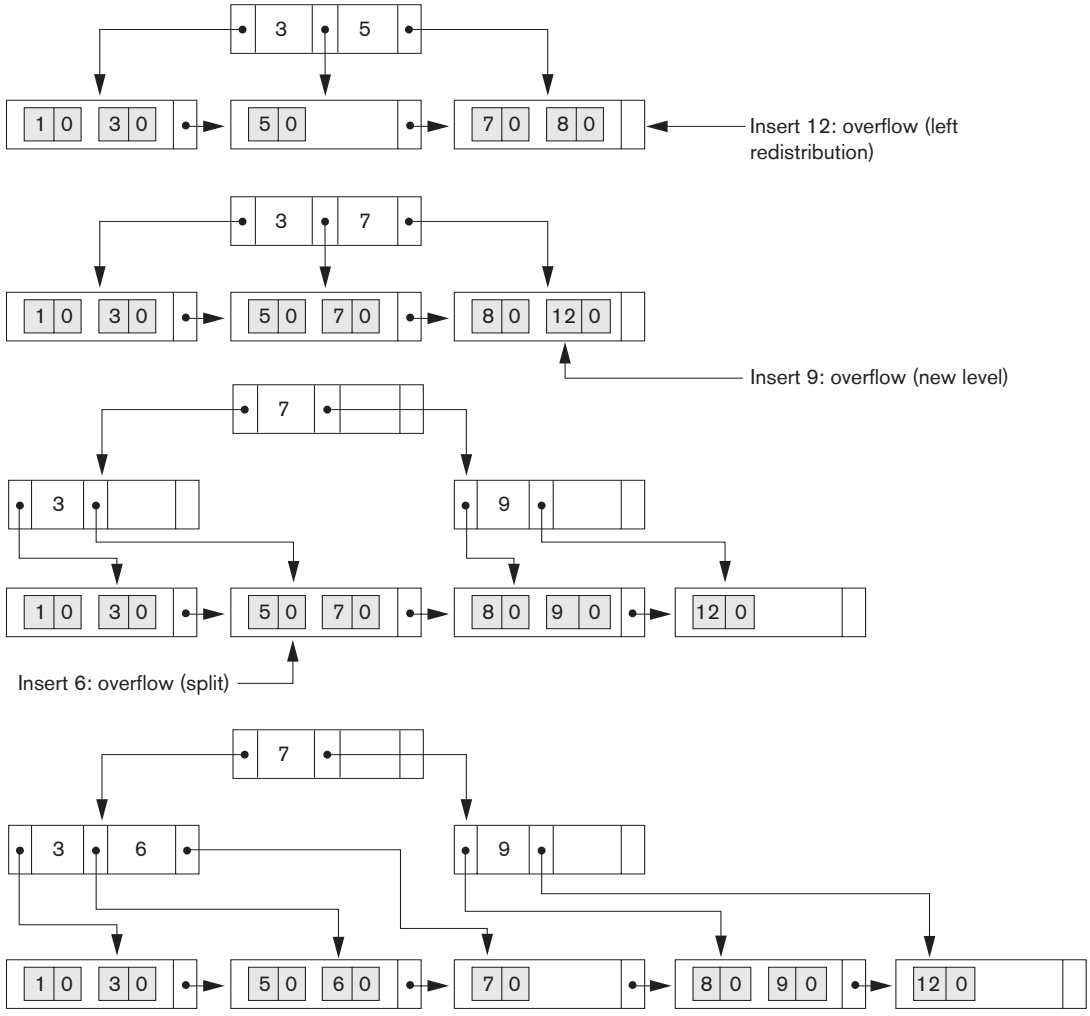


Figure 18.17
B^+-tree insertion with left redistribution.

This page intentionally left blank

part 8

Query Processing and Optimization, and Database Tuning

This page intentionally left blank

chapter 19

Algorithms for Query Processing and Optimization

In this chapter we discuss the techniques used internally by a DBMS to process, optimize, and execute high-level queries. A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.[1] 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**. The DBMS must then devise an **execution strategy** or **query plan** for retrieving the results of the query from the database files. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.

Figure 19.1 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.

[1]We will not discuss the parsing and syntax-checking phase of query processing here; this material is discussed in compiler textbooks.

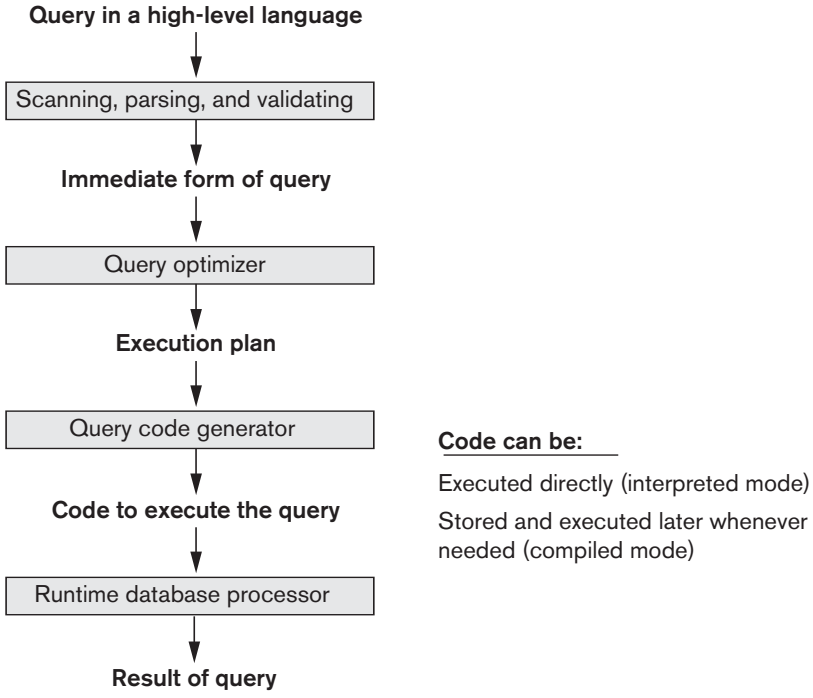


Figure 19.1
Typical steps when processing a high-level query.

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 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 may require detailed information on how the files are implemented and even on the contents of the files—information that may not be fully available in the DBMS catalog. 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 (see Section 2.6)—the programmer must choose the query execution strategy while writing a database program. If a DBMS provides only a navigational language, there is *limited need or 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 11) 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 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.[2] A relational DBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or near-optimal strategy. Each DBMS typically has a number of general database access algorithms that implement relational algebra operations such as SELECT or JOIN (see Chapter 6) 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 starts with a general discussion of how SQL queries are typically translated into relational algebra queries and then optimized in Section 19.1. Then we discuss algorithms for implementing relational algebra operations in Sections 19.2 through 19.6. Following this, we give an overview of query optimization strategies. There are two main techniques that are employed during query optimization. The first technique is based on **heuristic rules** for ordering the operations in a query execution strategy. A heuristic is a rule 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 **systematically estimating** the cost of different execution strategies and choosing the execution plan with the lowest cost estimate. These techniques are usually combined in a query optimizer. We discuss heuristic optimization in Section 19.7 and cost estimation in Section 19.8. Then we provide a brief overview of the factors considered during query optimization in the Oracle commercial RDBMS in Section 19.9. Section 19.10 introduces the topic of semantic query optimization, in which known constraints are used as an aid to devising efficient query execution strategies.

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 4 and 5), relational algebra (Chapter 6), and file structures and indexing (Chapters 17 and 18) 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 chapter.

19.1 Translating SQL Queries into Relational Algebra

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 and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as

[2]There are some query optimization problems and techniques that are pertinent only to ODBMSs. However, we do not discuss them here because we give only an introduction to query optimization.

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 6.4.

Consider the following SQL query on the EMPLOYEE relation in Figure 3.5:

```
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:

$$\Im_{\text{MAX Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$$

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 query (without correlation with the outer query)* in Section 5.1.2. It is much harder to optimize the more complex *correlated nested queries* (see Section 5.1.3), where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block.

19.2 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 18)—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.[3] 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 19.2, 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 $b = 1024$ 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 $\lceil (\log_{dM}(n_R)) \rceil$. 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.

[3] *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 books, and include techniques such as quick sort, heap sort, bubble sort, and many others. We do not discuss these here.

```
set     i ← 1;
        j ← b;              {size of the file in blocks}
        k ← n_B;            {size of buffer in blocks}
        m ← ⌈(j/k)⌉;

{Sorting Phase}
while (i ≤ 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 ← i + 1;
}

{Merging Phase: merge subfiles until only 1 remains}
set     i ← 1;
        p ← ⌈log_{k−1} m⌉   {p is the number of passes for the merging phase}
        j ← m;
while (i ≤ p)
do {
        n ← 1;
        q ← ⌈j/(k−1)⌉;  {number of subfiles to write in this pass}
        while (n ≤ 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 ← n + 1;
        }
        j ← q;
        i ← i + 1;
}
```

Figure 19.2
Outline of the sort-merge algorithm for external sorting.

The performance of the sort-merge algorithm can be measured in 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 6).

19.3 Algorithms for SELECT and JOIN Operations

19.3.1 Implementing 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 3.5, to illustrate our discussion:

OP1: $\sigma_{\text{Ssn} = \text{'123456789'}} (\text{EMPLOYEE})$

OP2: $\sigma_{\text{Dnumber} > 5} (\text{DEPARTMENT})$

OP3: $\sigma_{\text{Dno} = 5} (\text{EMPLOYEE})$

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

OP5: $\sigma_{\text{Essn='123456789' AND Pno} = 10} (\text{WORKS\_ON})$

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.[4] 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 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.

[4]A selection operation is sometimes called a **filter**, since it filters out the records in the file that do *not* satisfy the selection condition.