1. What is a communication protocol?
2. "What is the structure of an HTTP request message? What is the structure of an HTTP response message? Why do HTTP messages carry a version field?
3. What is a stateless protocol? Why was HTTP designed to be stateless?
4. Show the HTTP request message generated when you request the home page of this book (`http://www.cs.wisc.edu/~dbbook`). Show the HTTP response message that the server generates for that page.

Exercise 7.3 In this exercise, you are asked to write the functionality of a generic shopping basket; you will use this in several subsequent project exercises. Write a set of JSP pages that displays a shopping basket of items and allows users to add, remove, and change the quantity of items. To do this, use a cookie storage scheme that stores the following information:

- The UserId of the user who owns the shopping basket.
- The number of products stored in the shopping basket.
- A product id and a quantity for each product.

When manipulating cookies, remember to set the Expires property such that the cookie can persist for a session or indefinitely. Experiment with cookies using JSP and make sure you know how to retrieve, set values, and delete the cookie.

You need to create five JSP pages to make your prototype complete:

- Index Page (index.jsp): This is the main entry point. It has a link that directs the user to the Products page so they can start shopping.
- Products Page (products.jsp): Shows a listing of all products in the database with their descriptions and prices. This is the main page where the user fills out the shopping basket. Each listed product should have a button next to it, which adds it to the shopping basket. (If the item is already in the shopping basket, it increments the quantity by one.) There should also be a counter to show the total number of items currently in the shopping basket. Note that if a user has a quantity of five of a single item in the shopping basket, the counter should indicate a total quantity of five. The page also contains a button that directs the user to the Cart page.
- Cart Page (cart.jsp): Shows a listing of all items in the shopping basket cookie. The listing for each item should include the product name, price, a text box for the quantity (the user can change the quantity of items here), and a button to remove the item from the shopping basket. This page has three other buttons: one button to continue shopping (which returns the user to the Products page), a second button to update the cookie with the altered quantities from the text boxes, and a third button to place or confirm the order, which directs the user to the Confirm page.
- Confirm Page (confirm.jsp): Lists the final order. There are two buttons on this page. One button cancels the order and the other submits the completed order. The cancel button just deletes the cookie and returns the user to the Index page. The submit button updates the database with the new order, deletes the cookie, and returns the user to the Index page.

Exercise 7.4 In the previous exercise, replace the page products.jsp with the following *search page* search.jsp. This page allows users to search products by name or description. There should be both a text box for the search text and radio buttons to allow the

user to choose between search-by-name and search-by-description (as well as a submit button to retrieve the results). The page that handles search results should be modeled after products.jsp (as described in the previous exercise) and be called products.jsp. It should retrieve all records where the search text is a substring of the name or description (as chosen by the user). To integrate this with the previous exercise, simply replace all the links to products.jsp with search.jsp.

Exercise 7.5 'Write a simple authentication mechanism (without using encrypted transfer of passwords, for simplicity). We say a user is authenticated if she has provided a valid username-password combination to the system; otherwise, we say the user is not authenticated. Assume for simplicity that you have a database schema that stores only a customer id and a password:

Passwords(cid: integer, username: string, password: string)

1. How and where are you going to track when a user is 'logged on' to the system?
2. Design a page that allows a registered user to log on to the system.
3. Design a page header that checks whether the user visiting this page is logged in.

Exercise 7.6 (Due to Jeff Derstadt) TechnoBooks.com is in the process of reorganizing its website. A major issue is how to efficiently handle a large number of search results. In a human interaction study, it found that modem users typically like to view 20 search results at a time, and it would like to program this logic into the system. Queries that return batches of sorted results are called *top N queries*. (See Section 25.5 for a discussion of database support for top N queries.) For example, results 1-20 are returned, then results 21-40, then 41-60, and so on. Different techniques are used for performing top N queries and TechnoBooks.com would like you to implement two of them.

Infrastructure: Create a database with a table called Books and populate it with some books, using the format that follows. This gives you 111 books in your database with a title of AAA, BBB, CCC, DDD, or EEE, but the keys are not sequential for books with the same title.

Books(*bookid:* INTEGER, *title:* CHAR(80), *author:* CHAR(80), *price:* REAL)

```
For i = 1 to 111 {
        Insert the tuple (i, "AAA", "AAA Author", 5.99)
        i=i+1
        Insert the tuple (i, "BBB", "BBB Author", 5.99)
        i = i + 1
        Insert the tuple (i, "CCC", "CCC Author", 5.99)
        i=i+1
        Insert the tuple (i, "DDD", "DDD Author", 5.99)
        1=i+1
        Insert the tuple (i, "EEE", "EEE Author", 5.99)
```

Placeholder Technique: The simplest approach to top N queries is to store a placeholder for the first and last result tuples, and then perform the same query. When the new query results are returned, you can iterate to the placeholders and return the previous or next 20 results.

Tuples Shown	Lower Placeholder	Previous Set	Upper Placeholder	Next Set
1-20	1	None	20	21-40
21-40	21	1-20	40	41-60
41-60	41	21-40	60	61-80

Write a webpage in JSP that displays the contents of the Books table, sorted by the Title and BookId, and showing the results 20 at a time. There should be a link (where appropriate) to get the previous 20 results or the next 20 results. To do this, you can encode the placeholders in the Previous or Next Links as follows. Assume that you are displaying records 21-40. Then the previous link is display.jsp?lower=21 and the next link is display.jsp?upper=40.

You should not display a previous link when there are no previous results; nor should you show a Next link if there are no more results. When your page is called again to get another batch of results, you can perform the same query to get all the records, iterate through the result set until you are at the proper starting point, then display 20 more results.

What are the advantages and disadvantages of this technique?

Query Constraints Technique: A second technique for performing top N queries is to push boundary constraints into the query (in the WHERE clause) so that the query returns only results that have not yet been displayed. Although this changes the query, fewer results are returned and it saves the cost of iterating up to the boundary. For example, consider the following table, sorted by (title, primary key).

Batch	Result Number	Title	Primary Key
1	1	AAA	105
1	2	BBB	13
1	3	eee	48
1	4	DDD	52
1	5	DDD	101
2	6	DDD	121
2	7	EEE	19
2	8	EEE	68
2	9	FFF	2
2	10	FFF	33
3	11	FFF	58
3	12	FFF	59
3	13	GGG	93
3	14	EHH	132
3	15	HHH	135

In batch 1, rows 1 through 5 are displayed, in batch 2 rows 6 through 10 are displayed, and so on. Using the placeholder technique, all 15 results would be returned for each batch. Using the constraint technique, batch 1 displays results 1-5 but returns results 1-15, batch 2 will display results 6-10 but returns only results 6-15, and batch 3 will display results 11-15 but return only results 11-15.

The constraint can be pushed into the query because of the sorting of this table. Consider the following query for batch 2 (displaying results 6-10):

```
EXEC SQL SELECT B.Title
FROM     Books B
WHERE    (B.Title = 'DDD' AND B.BookId > 101) OR (B.Title > 'DDD')
ORDER BY B.Title, B.Bookld
```

This query first selects all books with the title 'DDD,' but with a primary key that is greater than that of record 5 (record 5 has a primary key of 101). This returns record 6. Also, any book that has a title after 'DDD' alphabetically is returned. You can then display the first five results.

The following information needs to be retained to have Previous and Next buttons that return more results:

- Previous: The title of the *first* record in the previous set, and the primary key of the *first* record in the previous set.
- Next: The title of the *first* record in the next set; the primary key of the *first* record in the next set.

These four pieces of information can be encoded into the Previous and Next buttons as in the previous part. Using your database table from the first part, write a JavaServer Page that displays the book information 20 records at a time. The page should include *Previous* and *Next* buttons to show the previous or next record set if there is one. Use the constraint query to get the Previous and Next record sets.

PROJECT-BASED EXERCISES

In this chapter, you continue the exercises from the previous chapter and create the parts of the application that reside at the middle tier and at the presentation tier. More information about these exercises and material for more exercises can be found online at

`http://www.cs.wisc.edu/~dbbook`

Exercise 7.7 Recall the Notown Records website that you worked on in Exercise 6.6. Next, you are asked to develop the actual pages for the Notown Records website. Design the part of the website that involves the presentation tier and the middle tier, and integrate the code that you wrote in Exercise 6.6 to access the database.

I. Describe in detail the set of webpages that users can access. Keep the following issues in mind:

   - All users start at a common page.
   - For each action, what input does the user provide? How will the user provide it -by clicking on a link or through an HTML form?
   - What sequence of steps does a user go through to purchase a record? Describe the high-level application flow by showing how each IIser action is handled.

2. Write the webpages in HTML without dynamic content.
3. Write a page that allows users to log on to the site. Use cookies to store the information permanently at the user's browser.
4. Augment the log-on page with JavaScript code that checks that the username consists only of the characters from a to z.
5. Augment the pages that allow users to store items in a shopping basket with a condition that checks whether the user has logged on to the site. If the user has not yet logged on, there should be no way to add items to the shopping cart. Implement this functionality using JSP by checking cookie information from the user.
6. Create the remaining pages to finish the website.

Exercise **7.8** Recall the online pharmacy project that you worked on in Exercise 6.7 in Chapter 6. Follow the analogous steps from Exercise 7.7 to design the application logic and presentation layer and finish the website.

Exercise **7.9** Recall the university database project that you worked on in Exercise 6.8 in Chapter 6. Follow the analogous steps from Exercise 7.7 to design the application logic and presentation layer and finish the website.

Exercise **7.10** Recall the airline reservation project that you worked on in Exercise 6.9 in Chapter 6. Follow the analogous steps from Exercise 7.7 to design the application logic and presentation layer and finish the website.

BIBLIOGRAPHIC NOTES

The latest version of the standards mentioned in this chapter can be found at the website of the World Wide Web Consortium (www.w3.org). It contains links to information about I-ITML, cascading style sheets, XIvIL, XSL, and much more. The book by Hall is a general introduction to Web programming technologies [357]; a good starting point on the Web is www.Webdeve1oper.com. There are many introductory books on CGI prograruning, for example [210, 198]. The JavaSoft (java.sun.com) home page is a good starting point for Servlets, .JSP, and all other Java-related technologies. The book by Hunter [394] is a good introduction to Java Servlets. Microsoft supports Active Server Pages (ASP), a comparable tedmology to .lSI'. More information about ASP can be found on the Microsoft Developer's Network horne page (msdn.microsoft.com).

There are excellent websites devoted to the advancement of XML, for example www.xm1.com and www.ibm.com/xm1. that also contain a plethora of links with information about the other standards. There are good introductory books on many diflerent aspects of XML, for example [195, 158,597,474, 381, 320]. Information about UNICODE can be found on its home page http://www.unicode.org.

Inforrnation about .lavaServer Pages ane! servlets can be found on the JavaSoft home page at java.sun.com at java.sun.com/products/jsp and at java.sun.com/products/servlet.

PART III

STORAGE AND INDEXING

8

OVERVIEW'OF STORAGE AND INDEXING

- How does a DBMS store and access persistent data?
- Why is I/O cost so important for database operations?
- How does a DBMS organize files of data records on disk to minimize I/O costs?
- What is an index, and why is it used?
- What is the relationship between a file of data records and any indexes on this file of records?
- What are important properties of indexes?
- How does a hash-based index work, and when is it most effective?
- How does a tree-based index work, and when is it most effective?
- How can we use indexes to optimize performance for a given workload?
- Key concepts: external storage, buffer manager, page I/O; file organization, heap files, sorted files; indexes, data entries, search keys, clustered index, clustered file, primary index; index organization, hash-based and tree-based indexes; cost comparison, file organizations and common operations; performance tuning, workload, composite search keys, use of clustering,

If you don't find it in the index, look very carefully through the entire catalog.

--Sears, Roebuck, and Co., Consumers' Guide, 1897

The basic abstraction of data in a DBMS is a collection of records, or a *file,* and each file consists of one or more pages. The *files and access methods*

software layer organizes data carefully to support fast access to desired subsets of records. Understanding how records are organized is essential to using a database system effectively, and it is the main topic of this chapter.

A file organization is a method of arranging the records in a file when the file is stored on disk. Each file organization makes certain operations efficient but other operations expensive.

Consider a file of employee records, each containing *age, name,* and *sal* fields, which we use as a running example in this chapter. If we want to retrieve employee records in order of increasing age, sorting the file by age is a good file organization, but the sort order is expensive to maintain if the file is frequently modified. Further, we are often interested in supporting more than one operation on a given collection of records. In our example, we may also want to retrieve all employees who make more than $5000. We have to scan the entire file to find such employee records.

A technique called *indexing* can help when we have to access a collection of records in multiple ways, in addition to efficiently supporting various kinds of selection. Section 8.2 introduces indexing, an important aspect of file organization in a DBMS. We present an overview of index data structures in Section 8.3; a more detailed discussion is included in Chapters 10 and 11.

We illustrate the importance of choosing an appropriate file organization in Section 8.4 through a simplified analysis of several alternative file organizations. The cost model used in this analysis, presented in Section 8.4.1, is used in later chapters as well. In Section 8.5, we highlight some important choices to be made in creating indexes. Choosing a good collection of indexes to build is arguably the single most powerful tool a database administrator has for improving performance.

8.1 DATA ON EXTERNAL STORAGE

A DBMS stores vast quantities of data, and the data must persist across program executions. Therefore, data is stored on external storage devices such as disks and tapes, and fetched into main memory as needed for processing. The unit of information read from or written to disk is a *page*. The size of a page is a DBMS parameter, and typical values are 4KB or 8KB.

The cost of page I/O (*input* from disk to main Inemory and *output* from memory to disk) dominates the cost of typical database operations, and database systems are carefully optimized to rninimize this cost. While the details of how

files of records are physically stored on disk and how main memory is utilized are covered in Chapter 9, the following points are important to keep in mind:

- Disks are the most important external storage devices. They allow us to retrieve any page at a (more or less) fixed cost per page. However, if we read several pages in the order that they are stored physically, the cost can be much less than the cost of reading the same pages in a random order.
- Tapes are sequential access devices and force us to read data one page after the other. They are mostly used to archive data that is not needed on a regular basis.
- Each record in a file has a unique identifier called a record id, or **rid** for short. An rid has the property that we can identify the disk address of the page containing the record by using the rid.

Data is read into memory for processing, and written to disk for persistent storage, by a layer of software called the *buffer manager*. When the *files and access methods* layer (which we often refer to as just the file layer) needs to process a page, it asks the buffer manager to fetch the page, specifying the page's rid. The buffer manager fetches the page from disk if it is not already in memory.

Space on disk is managed by the *disk space m,anager,* according to the DBMS software architecture described in Section 1.8. When the files and access methods layer needs additional space to hold new records in a file, it asks the disk space manager to allocate an additional disk page for the file; it also informs the disk space manager when it no longer needs one of its disk pages. The disk space manager keeps track of the pages in use by the file layer; if a page is freed by the file layer, the space rnanager tracks this, and reuses the space if the file layer requests a new page later on.

In the rest of this chapter, we focus on the files and access methods layer.

8.2 FILE ORGANIZATIONS AND INDEXING

The file of records is an important abstraction in a DBMS, and is implemented by the files and access methods layer of the code. A file can be created, destroyed, and have records inserted into and deleted from it. It also supports scallS; a scan operation allows us to step through all the records in the file one at a time. A relatioll is typically stored as a file of records.

The file layer stores the records in a file in a collection of disk pages. It keeps track of pages allocated to each file, and as records are inserted into and deleted from the file, it also tracks availa.ble space within pages allocated to the file.

The simplest file structure is an unordered file, or heap file. Records in a heap file are stored in random order across the pages of the file. A heap file organization supports retrieval of all records, or retrieval of a particular record specified by its rid; the file manager must keep track of the pages allocated for the file. (We defer the details of how a heap file is implemented to Chapter 9.)

An index is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations. An index allows us to efficiently retrieve all records that satisfy search conditions on the search key fields of the index. We can also create additional indexes on a given collection of data records, each with a different search key, to speed up search operations that are not efficiently supported by the file organization used to store the data records.

Consider our example of employee records. We can store the records in a file organized as an index on employee age; this is an alternative to sorting the file by age. Additionally, we can create an auxiliary index file based on salary, to speed up queries involving salary. The first file contains employee records, and the second contains records that allow us to locate employee records satisfying a query on salary.

We use the term data entry to refer to the records stored in an index file. A data entry with search key value k, denoted as $k*$, contains enough information to locate (one or more) data records with search key value k. We can efficiently search an index to find the desired data entries, and then use these to obtain data records (if these are distinct from data entries).

There are three main alternatives for what to store as a data entry in an index:

1. A data entry $k*$ is an actual data record (with search key value *k*).
2. A data entry is a *(k, rid)* pair, where *rid* is the record id of a data record with search key value *k*.
3. A data entry is a $\langle k, rid\text{-}list \rangle$ pair, where *rid-list* is a list of record ids of data records with search key value *k*.

Of course, if the index is used to store actual data records, Alternative (1), each entry $k*$ is a data record with search key value *k*. We can think of such an index as a special file organization. Such an indexed file organization can be used instead of, for exarnple, a sorted file or an unordered file of records.

Alternatives (2) and (3), which contain data entries that point to data records, are independent of the file organization that is used for the indexed file (i.e.,

the file that contains the data records). Alternative (3) offers better space utilization than Alternative (2), but data entries are variable in length, depending on the number of data records with a given search key value.

If we want to build more than one index on a collection of data records-for example, we want to build indexes on both the *age* and the *sal* fields for a collection of employee records—at most one of the indexes should use Alternative (1) because we should avoid storing data records multiple times.

8.2.1 Clustered Indexes

When a file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index, we say that the index is clustered; otherwise, it clustered is an unclustered index. An index that uses Alternative (1) is clustered, by definition. An index that uses Alternative (2) or (3) can be a clustered index only if the data records are sorted on the search key field. Otherwise, the order of the data records is random, defined purely by their physical order, and there is no reasonable way to arrange the data entries in the index in the same order.

In practice, files are rarely kept sorted since this is too expensive to maintain when the data is updated. So, in practice, a clustered index is an index that uses Alternative (1), and indexes that use Alternatives (2) or (3) are unclustered. We sometimes refer to an index using Alternative (1) as a clustered file, because the data entries are actual data records, and the index is therefore a file of data records. (As observed earlier, searches and scans on an index return only its data entries, even if it contains additional information to organize the data entries.)

The cost of using an index to answer a range search query can vary tremendously based on whether the index is clustered. If the index is clustered, i.e., we are using the search key of a clustered file, the rids in qualifying data entries point to a contiguous collection of records, and we need to retrieve only a few data pages. If the index is unclustered, each qualifying data entry could contain a rid that points to a distinct data page, leading to as many data page I/Os as the number of data entries that match the range selection, as illustrated in Figure 8.1. This point is discussed further in Chapter 13.

8.2.2 Primary and Secondary Indexes

An index on a set of fields that includes the *primary key* (see Chapter 3) is called a primary index; other indexes are called secondary indexes. (The terms *primary inde.T* and *secondaTy index* are sometimes used with a different

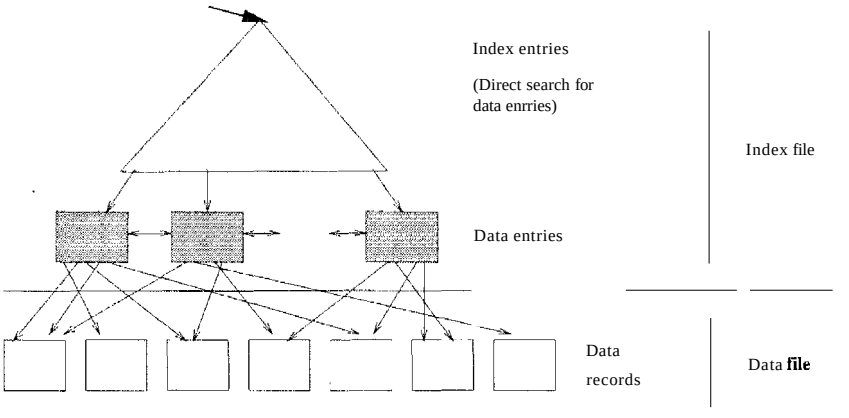


Figure 8.1 Uuelllst.ered Index Using Alternative (2)

meaning: An index that uses Alternative (1) is called a *primary index,* and one that uses Alternatives (2) or (3) is called a *secondary index.* We will be consistent with the definitions presented earlier, but the reader should be aware of this lack of standard terminology in the literature.)

Two data entries are said to be duplicates if they have the same value for the search key field associated with the index. A primary index is guaranteed not to contain duplicates, but an index on other (collections of) fields can contain duplicates. In general, a secondary index contains duplicates. If we know tha.t no duplicates exist, that is, we know that the search key contains some candidate key, we call the index a **unique** index.

An important issue is how data entries in an index are organized to support efficient retrieval of data entries.vVe discuss this next.

8.3 INDEX DATA STRUCTURES

One way to organize data entries is to hash data entries on the sea.rch key. Another way to organize data entries is to build a tree-like data structure that directs a search for data entries. We introduce these two basic approaches ill this section. We study tree-based indexing in more detail in Chapter 10 and hash-based indexing in Chapter 11.

We note that the choice of hash or tree indexing techniques can be combined with any of the three alternatives for data entries.

8.3.1 Hash-Based Indexing

We can organize records using a technique called *hashing* to quickly find records that have a given search key value. For example, if the file of employee records is hashed on the *name* field, we can retrieve all records about Joe.

In this approach, the records in a file are grouped in buckets, where a bucket consists of a primary page and, possibly, additional pages linked in a chain. The bucket to which a record belongs can be determined by applying a special function, called a hash function, to the search key. Given a bucket number, a hash-based index structure allows us to retrieve the primary page for the bucket in one or two disk l/Os.

On inserts, the record is inserted into the appropriate bucket, with 'overflow' pages allocated as necessary. To search for a record with a given search key value, we apply the hash function to identify the bucket to which such records belong and look at all pages in that bucket. If we do not have the search key value for the record, for example, the index is based on *sal* and we want records with a given age value, we have to scan all pages in the file.

In this chapter, we assume that applying the hash function to (the search key of) a record allows us to identify and retrieve the page containing the record with one I/O. In practice, hash-based index structures that adjust gracefully to inserts and deletes and allow us to retrieve the page containing a record in one to two l/Os (see Chapter 11) are known.

Hash indexing is illustrated in Figure 8.2, where the data is stored in a file that is hashed on *age;* the data entries in this first index file are the actual data records. Applying the hash function to the age field identifies the page that the record belongs to. The hash function *h* for this example is quite simple; it converts the search key value to its binary representation and uses the two least significant bits as the bucket identifier.

Figure 8.2 also shows an index with search key *sal* that contains *(sal, rid)* pairs as data entries. The *rid* (short for *record id)* component of a data entry in this second index is a pointer to a record with search key value *sal* (and is shown in the figure as an arrow pointing to the data record).

Using the terminology introduced in Section 8.2, Figure 8.2 illustrates Alternatives (1) and (2) for data entries. The file of employee records is hashed on *age,* and Alternative (1) is used for for data entries. The second index, on *sal,* also uses hashing to locate data entries, which are now *(sal, rid of employee record)* pairs; that is, Alternative (2) is used for data entries.

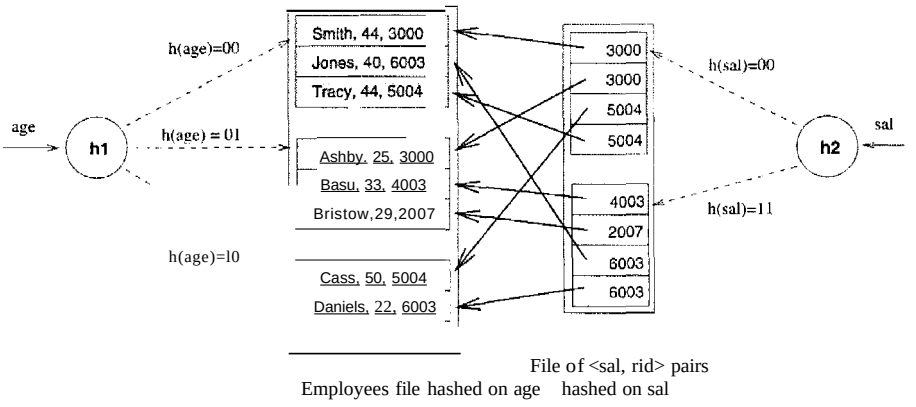


Figure 8.2 Index-Organized File Hashed on *age,* with Auxiliary Index on *sal*

Note that the search key for an index can be any sequence of one or more fields, and it need not uniquely identify records. For example, in the salary index, two data entries have the same search key value 6003. (There is an unfortunate overloading of the term *key* in the database literature. A *primary key* or *candidate* key-fields that uniquely identify a record; see Chapter 3—is unrelated to the concept of a search key.)

8.3.2 Tree-Based Indexing

An alternative to hash-based indexing is to organize records using a tree-like data structure. The data entries are arranged in sorted order by search key value, and a hierarchical search data structure is maintained that directs searches to the correct page of data entries.

Figure 8.3 shows the employee records from Figure 8.2, this time organized in a tree-structured index with search *keyage*. Each node in this figure (e.g., nodes labeled A, B, L1, L2) is a physical page, and retrieving a node involves a disk I/O.

The lowest level of the tree, called the **leaf level,** contains the data entries; in our example, these are employee records. To illustrate the ideas better, we have drawn Figure 8.3 as if there were additional employee records, some with age less than 22 and some with age greater than 50 (the lowest and highest age values that appear in Figure 8.2). Additional records with age less than 22 would appear in leaf pages to the left page L1, and records with age greater than 50 would appear in leaf pages to the right of page L3.

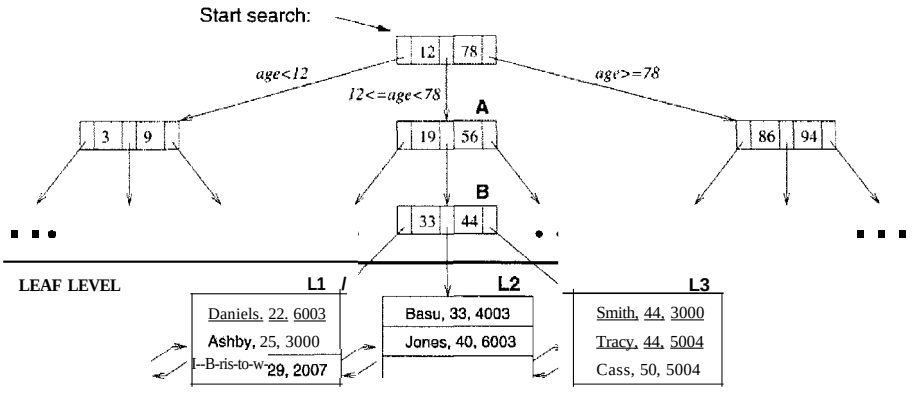


Figure 8.3 Tree·Structured Index

This structure allows us to efficiently locate all data entries with search key values in a desired range. All searches begin at the topmost node, called the root, and the contents of pages in non-leaf levels direct searches to the correct leaf page. Non-leaf pages contain node pointers separated by search key values. The node pointer to the left of a key value *k* points to a subtree that contains only data entries less than *k*. The node pointer to the right of a key value *k* points to a subtree that contains only data entries greater than or equal to *k*.

In our example, suppose we want to find all data entries with $24 < age < 50$. Each edge from the root node to a child node in Figure 8.2 has a label that explains what the corresponding subtree contains. (Although the labels for the remaining edges in the figure are not shown, they should be easy to deduce.) In our example search, we look for data entries with search key value > 24, and get directed to the middle child, node A. Again, examining the contents of this node, we are directed to node B. Examining the contents of node B, we are directed to leaf node L1, which contains data entries we are looking for.

Observe that leaf nodes L2 and L3 also contain data entries that satisfy our search criterion. To facilitate retrieval of such qualifying entries during search, all leaf pages are maintained in a doubly-linked list. Thus, we can fetch page L2 using the 'next' pointer on page L1, and then fetch page L3 using the 'next' pointer on L2.

Thus, the number of disk I/Os incurred during a search is equal to the length of a path from the root to a leaf, plus the number of leaf pages with qualifying data entries. The B+ tree is an index structure that ensures that all paths from the root to a leaf in a given tree are of the same length, that is, the structure is always balanced in height. Finding the correct leaf page is faster

than binary search of the pages in a sorted file because each non-leaf node can accommodate a very large number of node-pointers, and the height of the tree is rarely more than three or four in practice. The **height** of a balanced tree is the length of a path from root to leaf; in Figure 8.3, the height is three. The number of I/Os to retrieve a desired leaf page is four, including the root and the leaf page. (In practice, the root is typically in the buffer pool because it is frequently accessed, and we really incur just three I/Os for a tree of height three.)

The average number of children for a non-leaf node is called the **fan-out** of the tree. If every non-leaf node has n children, a tree of height h has n^h leaf pages. In practice, nodes do not have the same number of children, but using the average value F for n, we still get a good approximation to the number of leaf pages, F^h. In practice, F is at least 100, which means a tree of height four contains 100 million leaf pages. Thus, we can search a file with 100 million leaf pages and find the page we want using four I/Os; in contrast, binary search of the same file would take $\log_2 100{,}000{,}000$ (over 25) I/Os.

8.4 COMPARISON OF FILE ORGANIZATIONS

We now compare the costs of some simple operations for several basic file organizations on a collection of employee records. We assume that the files and indexes are organized according to the composite search key $\langle age, sal \rangle$, and that all selection operations are specified on these fields. The organizations that we consider are the following:

- File of randomly ordered employee records, or heap file.
- File of employee records sorted on *(age, sal)*.
- Clustered B+ tree file with search key *(age, sal)*.
- Heap file with an unclustered B+ tree index on *(age, sal)*.
- Heap file with an unclustered hash index on *(age, sal)*.

Our goal is to emphasize the importance of the choice of an appropriate file organization, and the above list includes the main alternatives to consider in practice. Obviously, we can keep the records unsorted or sort them. We can also choose to build an index on the data file. Note that even if the data file is sorted, an index whose search key differs from the sort order behaves like an index on a heap file!

The operations we consider are these:

- Scan: Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. There is also a CPU overhead per record for locating the record on the page (in the pool).
- Search with Equality Selection: Fetch all records that satisfy an equality selection; for example, "Find the employee record for the employee with *age* 23 and *sal* 50." Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.
- Search with Range Selection: Fetch all records that satisfy a range selection; for example, "Find all employee records with *age* greater than 35."
- Insert a Record: Insert a given record into the file. We must identify the page in the file into which the new record must be inserted, fetch that page from disk, modify it to include the new record, and then write back the modified page. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.
- Delete a Record: Delete a record that is specified using its rid. We must identify the page that contains the record, fetch it from disk, modify it, and write it back. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

8.4.1 Cost Model

In our comparison of file organizations, and in later chapters, we use a simple cost model that allows us to estimate the cost (in terms of execution time) of different database operations. We use B to denote the number of data pages when records are packed onto pages with no wasted space, and R to denote the number of records per page. The average time to read or write a disk page is D, and the average time to process a record (e.g., to compare a field value to a selection constant) is C. In the hashed file organization, we use a function, called a *hash function*, to map a record into a range of numbers; the time required to apply the hash function to a record is H. For tree indexes, we will use F to denote the fan-out, which typically is at least 100 as mentioned in Section 8.3.2.

Typical values today are D = 15 milliseconds, C and H = 100 nanoseconds; we therefore expect the cost of I/O to dominate. I/O is often (even typically) the dominant component of the cost of database operations, and so considering I/O costs gives us a good first approximation to the true costs. Further, CPU speeds are steadily rising, whereas disk speeds are not increasing at a similar pace. (On the other hand, as main memory sizes increase, a much larger fraction of the needed pages are likely to fit in memory, leading to fewer I/O requests!) We

have chosen to concentrate on the I/O component of the cost model, and we assume the simple constant *C* for in-memory per-record processing cost. Bear the follO\ving observations in mind:

- Real systems must consider other aspects of cost, such as CPU costs (and network transmission costs in a distributed database).
- Even with our decision to focus on I/O costs, an accurate model would be too complex for our purposes of conveying the essential ideas in a simple way. We therefore use a simplistic model in which we just count the number of pages read from or written to disk as a measure of I/O. We ignore the important issue of **blocked** access in our analysis-typically, disk systems allow us to read a block of contiguous pages in a single I/O request. The cost is equal to the time required to *seek* the first page in the block and transfer all pages in the block. Such blocked access can be much cheaper than issuing one I/O request per page in the block, especially if these requests do not follow consecutively, because we would have an additional seek cost for each page in the block.

We discuss the implications of the cost model whenever our simplifying assumptions are likely to affect our conclusions in an important way.

8.4.2 Heap Files

Scan: The cost is $B(D+RC)$ because we must retrieve each of B pages taking time D per page, and for each page, process R records taking time C per record.

Search with Equality Selection: Suppose that we know in advance that exactly one record matches the desired equality selection, that is, the selection is specified on a candidate key. On average, we must scan half the file, assuming that the record exists and the distribution of values in the search field is uniform. For each retrieved data page, we must check all records on the page to see if it is the desired record. The cost is $0.5B(D+RC)$. If no record satisfies the selection, however, we must scan the entire file to verify this.

If the selection is not on a candidate key field (e.g., "Find employees aged 18"), we always have to scan the entire file because records with *age* = 18 could be dispersed all over the file, and we have no idea how many such records exist.

Search with Range Selection: The entire file must be scanned because qualifying records could appear anywhere in the file, and we do not know how many qualifying records exist. The cost is $B(D+RC)$.

Insert: We assume that records are always inserted at the end of the file. We must fetch the last page in the file, add the record, and write the page back. The cost is $2D + C$.

Delete: We must find the record, remove the record from the page, and write the modified page back. We assume that no attempt is made to compact the file to reclaim the free space created by deletions, for simplicity.[1] The cost is the cost of searching plus $C + D$.

We assume that the record to be deleted is specified using the record id. Since the page id can easily be obtained from the record id, we can directly read in the page. The cost of searching is therefore D.

If the record to be deleted is specified using an equality or range condition on some fields, the cost of searching is given in our discussion of equality and range selections. The cost of deletion is also affected by the number of qualifying records, since all pages containing such records must be modified.

8.4.3 Sorted Files

Scan: The cost is $B(D + RC)$ because all pages must be examined. Note that this case is no better or worse than the case of unordered files. However, the order in which records are retrieved corresponds to the sort order, that is, all records in *age* order, and for a given age, by *sal* order.

Search with Equality Selection: We assume that the equality selection matches the sort order *(age, sal)*. In other words, we assume that a selection condition is specified on at least the first field in the composite key (e.g., *age* = 30). If not (e.g., selection *sal* = 50 or *department* = *"Toy"*), the sort order does not help us and the cost is identical to that for a heap file.

We can locate the first page containing the desired record or records, should any qualifying records exist, with a binary search in $log2B$ steps. (This analysis assumes that the pages in the sorted file are stored sequentially, and we can retrieve the ith page on the file directly in one disk I/O.) Each step requires a disk I/O and two cornparisons. Once the page is known, the first qualifying record can again be located by a binary search of the page at a cost of $Clog_2R$. The cost is $Dlog_2B + Clog_2R$, which is a significant improvement over searching heap files.

]In practice, a directory or other data structure is used to keep track of free space, and records are inserted into the first available free slot. as discussed in Chapter 9. This increases the cost of insertion and deletion a little, but not enough to affect our comparison.

If several records qualify (e.g., "Find all employees aged 18"), they are guaranteed to be adjacent to each other due to the sorting on *age*, and so the cost of retrieving all such records is the cost of locating the first such record $(D log_2 B + C log_2 R)$ plus the cost of reading all the qualifying records in sequential order. Typically, all qualifying records fit on a single page. If no records qualify, this is established by the search for the first qualifying record, which finds the page that would have contained a qualifying record, had one existed, and searches that page.

Search with Range Selection: Again assuming that the range selection matches the composite key, the first record that satisfies the selection is located as for search with equality. Subsequently, data pages are sequentially retrieved until a record is found that does not satisfy the range selection; this is similar to an equality search with many qualifying records.

The cost is the cost of search plus the cost of retrieving the set of records that satisfy the search. The cost of the search includes the cost of fetching the first page containing qualifying, or matching, records. For small range selections, all qualifying records appear on this page. For larger range selections, we have to fetch additional pages containing matching records.

Insert: To insert a record while preserving the sort order, we must first find the correct position in the file, add the record, and then fetch and rewrite all subsequent pages (because all the old records are shifted by one slot, assuming that the file has no empty slots). On average, we can assume that the inserted record belongs in the middle of the file. Therefore, we must read the latter half of the file and then write it back after adding the new record. The cost is that of searching to find the position of the new record plus $2 * (0.5B(D + RC))$, that is, search cost plus $B(D + RC)$.

Delete: We must search for the record, remove the record from the page, and write the modified page back. We must also read and write all subsequent pages because all records that follow the deleted record must be moved up to compact the free space.[2] The cost is the same as for an insert, that is, search cost plus $B(D + RC)$. Given the rid of the record to delete, we can fetch the page containing the record directly.

If records to be deleted are specified by an equality or range condition, the cost of deletion depends on the number of qualifying records. If the condition is specified on the sort field, qualifying records are guaranteed to be contiguous, and the first qualifying record can be located using binary search.

2Unlike a heap file, there is no inexpensive way to manage free space, so we account for the cost of compacting a file when a record is deleted.

8.4.4 Clustered Files

In a clustered file, extensive empirical study has shown that pages are usually at about 67 percent occupancy. Thus, the Humber of physical data pages is about *1.5B,* and we use this observation in the following analysis.

Scan: The cost of a scan is *1.5B(D + RC)* because all data pages must be examined; this is similar to sorted files, with the obvious adjustment for the increased number of data pages. Note that our cost metric does not capture potential differences in cost due to sequential I/O. We would expect sorted files to be superior in this regard, although a clustered file using ISAM (rather than B+ trees) would be close.

Search with Equality Selection: We assume that the equality selection matches the search key *(age, sal).* We can locate the first page containing the desired record or records, should any qualifying records exist, in *logF1.5B* steps, that is, by fetching all pages from the root to the appropriate leaf. In practice, the root page is likely to be in the buffer pool and we save an I/O, but we ignore this in our simplified analysis. Each step requires a disk I/O and two comparisons. Once the page is known, the first qualifying record can again be located by a binary search of the page at a cost of *Clog2R.* The cost is $DlogF1.5B + Clog_2R$, which is a significant improvement over searching even sorted files.

If several records qualify (e.g., "Find all employees aged 18"), they are guaranteed to be adjacent to each other due to the sorting on *age,* and so the cost of retrieving all such records is the cost of locating the first such record $(Dlogp1.5B + Clog_2R)$ plus the cost of reading all the qualifying records in sequential order.

Search with Range Selection: Again assuming that the range selection matches the composite key, the first record that satisfies the selection is located as it is for search with equality. Subsequently, data pages are sequentially retrieved (using the next and previous links at the leaf level) until a record is found that does not satisfy the range selection; this is similar to an equality search with many qualifying records.

Insert: To insert a record, we must first find the correct leaf page in the index, reading every page from root to leaf. Then, we must add the llew record. Most of the time, the leaf page has sufficient space for the new record, and all we need to do is to write out the modified leaf page. Occasionally, the leaf is full and we need to retrieve and modify other pages, but this is sufficiently rare

that we can ignore it in this simplified analysis. The cost is therefore the cost of search plus one write, $Dlog_F L5B + Clog_2 R + D$.

Delete: We must search for the record, remove the record from the page, and write the modified page back. The discussion and cost analysis for insert applies here as well.

8.4.5 Heap File with Unclustered Tree Index

The number of leaf pages in an index depends on the size of a data entry. We assume that each data entry in the index is a tenth the size of an employee data record, which is typical. The number of leaf pages in the index is o.1(L5B) = $O.15B$, if we take into account the 67 percent occupancy of index pages. Similarly, the number of data entries on a page $10(0.67R) = 6.7R$, taking into account the relative size and occupancy.

Scan: Consider Figure 8.1, which illustrates an unclustered index. To do a full scan of the file of employee records, we can scan the leaf level of the index and for each data entry, fetch the corresponding data record from the underlying file, obtaining data records in the sort order *(age, sal)*.

We can read all data entries at a cost of $O.15B(D + 6.7RC)$ l/Os. Now comes the expensive part: We have to fetch the employee record for each data entry in the index. The cost of fetching the employee records is one I/O per record, since the index is unclustered and each data entry on a leaf page of the index could point to a different page in the employee file. The cost of this step is $BR(D + C)$, which is prohibitively high. If we want the employee records in sorted order, we would be better off ignoring the index and scanning the employee file directly, and then sorting it. A simple rule of thumb is that a file can be sorted by a two-pass algorithm in which each pass requires reading and writing the entire file. Thus, the I/O cost of sorting a file with B pages is $4B$, which is much less than the cost of using an unclustered index.

Search with Equality Selection: We assume that the equality selection matches the sort order $\langle age, sal \rangle$. We can locate the first page containing the desired data entry or entries, should any qualifying entries exist, in $lagrO.15B$ steps, that is, by fetching all pages from the root to the appropriate leaf. Each step requires a disk I/O and two comparisons. Once the page is known, the first qua1ifying data entry can again be located by a binary search of the page at a cost of $Clog_2 6.7R$. The first qualifying data record can be fetched fronl the employee file with another I/O. The cost is $DlogpO.15B + Clag26.7R + D$, which is a significant improvement over searching sorted files.

If several records qualify (e.g., "Find all employees aged 18"), they are *not* guaranteed to be adjacent to each other. The cost of retrieving all such records is the cost oflocating the first qualifying data entry *(Dlo9pO.15B + Clo926.7R)* plus one I/O per qualifying record. The cost of using an unclustered index is therefore very dependent on the number of qualifying records.

Search with Range Selection: Again assuming that the range selection matches the composite key, the first record that satisfies the selection is located as it is for search with equality. Subsequently, data entries are sequentially retrieved (using the next and previous links at the leaf level of the index) until a data entry is found that does not satisfy the range selection. For each qualifying data entry, we incur one I/O to fetch the corresponding employee records. The cost can quickly become prohibitive as the number of records that satisfy the range selection increases. As a rule of thumb, if 10 percent of data records satisfy the selection condition, we are better off retrieving all employee records, sorting them, and then retaining those that satisfy the selection.

Insert: We must first insert the record in the employee heap file, at a cost of *2D* + C. In addition, we must insert the corresponding data entry in the index. Finding the right leaf page costs *Dl09pO.15B + Cl0926.7R*, and writing it out after adding the new data entry costs another *D.*

Delete: We need to locate the data record in the employee file and the data entry in the index, and this search step costs *Dl09FO.15B + Cl0926.7R + D*. Now, we need to write out the modified pages in the index and the data file, at a cost of *2D*.

8.4.6 Heap File With Unclustered Hash Index

As for unclustered tree indexes, we assume that each data entry is one tenth the size of a data record. We consider only static hashing in our analysis, and for simplicity we assume that there are no overflow chains.[3]

In a static hashed file, pages are kept at about SO percent occupancy (to leave space for future insertions and minimize overflows as the file expands). This is achieved by adding a new page to a bucket when each existing page is SO percent full, when records are initially loaded into a hashed file structure. The number of pages required to store data entries is therefore 1.2.5 times the number of pages when the entries are densely packed, that is, *1.25(0.10B) = O.125B*. The number of data entries that fit on a page is 1O(O.80R) = *8R*, taking into account the relative size and occupancy.

[3]The dynamic variants of hashing are less susceptible to the problem of overflow chains, and have a slight.ly higher average cost per search, but are otherwise similar to the static version.

Scan: As for an unclustered tree index, all data entries can be retrieved inexpensively, at a cost of O.125B(D + *8RC)* I/Os. However, for each entry, we incur the additional cost of one I/O to fetch the corresponding data record; the cost of this step is *BR(D + C)*. This is prohibitively expensive, and further, results are unordered. So no one ever scans a hash index.

Search with Equality Selection: This operation is supported very efficiently for matching selections, that is, equality conditions are specified for each field in the composite search key *(age, sal)*. The cost of identifying the page that contains qualifying data entries is *H*. Assuming that this bucket consists of just one page (i.e., no overflow pages), retrieving it costs *D*. If we assume that we find the data entry after scanning half the records on the page, the cost of scanning the page is *O.5(8R)C = 4RC*. Finally, we have to fetch the data record from the employee file, which is another *D*. The total cost is therefore *H + 2D + 4RC*, which is even lower than the cost for a tree index.

If several records qualify, they are *not* guaranteed to be adjacent to each other. The cost of retrieving all such records is the cost of locating the first qualifying data entry *(H + D + 4RC)* plus one I/O per qualifying record. The cost of using an unclustered index therefore depends heavily on the number of qualifying records.

Search with Range Selection: The hash structure offers no help, and the entire heap file of employee records must be scanned at a cost of *B(D + RC)*.

Insert: We must first insert the record in the employee heap file, at a cost of *2D* + C. In addition, the appropriate page in the index must be located, modified to insert a new data entry, and then written back. The additional cost is *H + 2D* + C.

Delete: We need to locate the data record in the employee file and the data entry in the index; this search step costs *H + 2D + 4RC*. Now, we need to write out the modified pages in the index and the data file, at a cost of *2D*.

8.4.7 Comparison of I/O Costs

Figure 8.4 compares I/O costs for the various file organizations that we discussed. A heap file has good storage efficiency and supports fast scanning and insertion of records. However, it is slow for searches and deletions.

A sorted file also offers good storage efficiency. but insertion and deletion of records is slow. Searches are fa.ster than in heap files. It is worth noting that, in a real DBMS, a file is almost never kept fully sorted.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap	BD	$0.5BD$	BD	$2D$	*Search+* D
Sorted	BD	$Dlog_2B$	*Dlog2B +# matching pages*	*Search* + BD	*Search+ BD*
Clustered	$1.5BD$	*DlogF1.5B*	*Dlo9F1.5B+# matching pages*	*Search* + D	*Search+* D
Unclustered tree index	$BD(R$ + 0.15)	D(1 + *logFO.15B)*	*D(lo9FO.15B+# matching records)*	*D(3* + *logFO.15B)*	*Search+* 2D
Unclustered hash index	$BD(R$ + 0.125)	2D	*BD*	4D	*Search+* 2D

Figure 8.4 A Comparison of I/O Costs

A clustered file offers all the advantages of a sorted file *and* supports inserts and deletes efficiently. (There is a space overhead for these benefits, relative to a sorted file, but the trade-off is well worth it.) Searches are even faster than in sorted files, although a sorted file can be faster when a large number of records are retrieved sequentially, because of blocked I/O efficiencies.

Unclustered tree and hash indexes offer fast searches, insertion, and deletion, but scans and range searches with many matches are slow. Hash indexes are a little faster on equality searches, but they do not support range searches.

In summary, Figure 8.4 demonstrates that no one file organization is uniformly superior in all situations.

8.5 INDEXES AND PERFORMANCE TUNING

In this section, we present an overview of choices that arise when using indexes to improve performance in a database system. The choice of indexes has a tremendous impact on system performance, and must be made in the context of the expected workload, or typical mix of queries and update operations.

A full discussion of indexes and performance requires an understanding of database query evaluation and concurrency control. We therefore return to this topic in Chapter 20, where we build on the discussion in this section. In particular, we discuss examples involving multiple tables in Chapter 20 because they require an understanding of join algorithms and query evaluation plans.

8.5.1 Impact of the Workload

The first thing to consider is the expected workload and the common operations. Different file organizations and indexes, as we have seen, support different operations well.

In generaL an index supports efficient retrieval of data entries that satisfy a given selection condition. Recall from the previous section that there are two important kinds of selections: equality selection and range selection. Hash-based indexing techniques are optimized only for equality selections and fare poorly on range selections. where they are typically worse than scanning the entire file of records. Tree-based indexing techniques support both kinds of selection conditions efficiently, explaining their widespread use.

Both tree and hash indexes can support inserts, deletes, and updates quite efficiently. Tree-based indexes, in particular, offer a superior alternative to maintaining fully sorted files of records. In contrast to simply maintaining the data entries in a sorted file, our discussion of (B+ tree) tree-structured indexes in Section 8.3.2 highlights two important advantages over sorted files:

1. We can handle inserts and deletes of data entries efficiently.
2. Finding the correct leaf page when searching for a record by search key value is much faster than binary search of the pages in a sorted file.

The one relative disadvantage is that the pages in a sorted file can be allocated in physical order on disk, making it much faster to retrieve several pages in sequential order. Of course. inserts and deletes on a sorted file are extremely expensive. A variant of B+ trees, called Indexed Sequential Access Method (ISAM), offers the benefit of sequential allocation of leaf pages, plus the benefit of fast searches. Inserts and deletes are not handled as well as in B+ trees, but are rnuch better than in a sorted file. \Ve will study tree-structured indexing in detail in Chapter 10.

8.5.2 Clustered Index Organization

As we saw in Section 8.2.1, a clustered index is really a file organization for the underlying data records. Data records can be la.rge, and we should avoid replicating them; so there can be at most one clustered index on a given collection of records. On the other hand, we can build several uncIustered indexes on a data file. Suppose that employee records are sorted by *age*, or stored in a clustered file with search keyage. If. in addition. we have an index on the *sal* field, the latter nlUst be an llnclllstered index. We can also build an uncIustered index on. say, *department*, if there is such a field.

Clustered indexes, while less expensive to maintain than a fully sorted file, are nonetheless expensive to maintain. When a new record has to be inserted into a full leaf page, a new leaf page must be allocated and sorne existing records have to be moved to the new page. If records are identified by a combination of page id and slot, as is typically the case in current database systems, all places in the database that point to a moved record (typically, entries in other indexes for the same collection of records) must also be updated to point to the new location. Locating all such places and making these additional updates can involve several disk I/Os. Clustering must be used sparingly and only when justified by frequent queries that benefit from clustering. In particular, there is no good reason to build a clustered file using hashing, since range queries cannot be answered using hash-indexes.

In dealing with the limitation that at most one index can be clustered, it is often useful to consider whether the information in an index's search key is sufficient to answer the query. If so, modern database systems are intelligent enough to avoid fetching the actual data records. For example, if we have an index on *age,* and we want to compute the average age of employees, the DBMS can do this by simply examining the data entries in the index. This is an example of an index-only evaluation. In an index-only evaluation of a query we need not access the data records in the files that contain the relations in the query; we can evaluate the query completely through indexes on the files. An important benefit of index-only evaluation is that it works equally efficiently with only unclustered indexes, as only the data entries of the index are used in the queries. Thus, unclustered indexes can be used to speed up certain queries if we recognize that the DBMS will exploit index-only evaluation.

Design Examples Illustrating Clustered Indexes

To illustrate the use of a clustered index 011 a range query, consider the following example:

```
SELECT    E.dno
FROM      Employees E
WHERE     E.age > 40
```

If we have a H+ tree index on *age,* we can use it to retrieve only tuples that satisfy the selection E.*age*> 40. Whether such an index is worthwhile depends first of all on the selectivity of the condition. What fraction of the employees are older than 40'1 If virtually everyone is older than 40, we gain little by using an index 011 *age*; a sequential scan of the relation would do almost as well. However, suppose that only 10 percent of the employees are older than 40. Now, is an index useful? The answer depends on whether the index is clustered. If the

index is unclustered, we could have one page I/O per qualifying employee, and this could be more expensive than a sequential scan, even if only 10 percent of the employees qualify! On the other hand, a clustered B+ tree index on *age* requires only 10 percent of the I/Os for a sequential scan (ignoring the few I/Os needed to traverse from the root to the first retrieved leaf page and the I/Os for the relevant index leaf pages).

As another example, consider the following refinement of the previous query:

```
SELECT    Kdno, COUNT(*)
FROM      Employees E
WHERE     E.age> 10
GROUP BY  E.dno
```

If a B+ tree index is available on *age,* we could retrieve tuples using it, sort the retrieved tuples on *dna,* and so answer the query. However, this may not be a good plan if virtually all employees are more than 10 years old. This plan is especially bad if the index is not clustered.

Let us consider whether an index on *dna* might suit our purposes better. We could use the index to retrieve all tuples, grouped by *dna,* and for each *dna* count the number of tuples with $age > 10$. (This strategy can be used with both hash and B+ tree indexes; we only require the tuples to be *grouped,* not necessarily *sorted,* by *dna.)* Again, the efficiency depends crucially on whether the index is clustered. If it is, this plan is likely to be the best if the condition on *age* is not very selective. (Even if we have a clustered index on *age,* if the condition on *age* is not selective, the cost of sorting qualifying tuples on *dna* is likely to be high.) If the index is not clustered, we could perform one page I/O per tuple in Employees, and this plan would be terrible. Indeed, if the index is not clustered, the optimizer will choose the straightforward plan based on sorting on *dna.* Therefore, this query suggests that we build a clustered index on *dna* if the condition on *age* is not very selective. If the condition is very selective, we should consider building an index (not necessarily clustered) on *age* instead.

Clustering is also important for an index on a search key that does not include a candidate key, that is, an index in which several data entries can have the same key value. To illustrate this point, we present the following query:

```
SELECT  E.dno
FROM    Employees E
WHERE   E.hobby='Stamps'
```

If many people collect stamps, retrieving tuples through an unclustered index on *hobby* can be very inefficient. It may be cheaper to simply scan the relation to retrieve all tuples and to apply the selection on-the-fly to the retrieved tuples. Therefore, if such a query is important, we should consider making the index on *hobby* a clustered index. On the other hand, if we assume that *eid* is a key for Employees, and replace the condition *E.hobby='Stamps'* by *E.eid=552,* we know that at most one Employees tuple will satisfy this selection condition. In this case, there is no advantage to making the index clustered.

The next query shows how aggregate operations can influence the choice of indexes:

```
SELECT    E.dno, COUNT(*)
FROM      Employees E
GROUP BY  E.dno
```

A straightforward plan for this query is to sort Employees on *dno* to compute the count of employees for each *dno.* However, if an index-hash or B+ tree---on *dno* is available, we can answer this query by scanning only the index. For each *dno* value, we simply count the number of data entries in the index with this value for the search key. Note that it does not matter whether the index is clustered because we never retrieve tuples of Employees.

8.5.3 Composite Search Keys

The search key for an index can contain several fields; such keys are called composite search keys or concatenated keys. As an example, consider a collection of employee records, with fields *name, age,* and *sal,* stored in sorted order by *name.* Figure 8.5 illustrates the difference between a composite index with key *(age, sal⟩*, a composite index with key *(sal, age),* an index with key *age,* and an index with key *sal.* All indexes shown in the figure use Alternative (2) for data entries.

If the search key is composite, an equality query is one in which *each* field in the search key is bound to a constant. For example, we can ask to retrieve all data entries with *age* = 20 and *sal* = 10. The hashed file organization supports only equality queries, since a hash function identifies the bucket containing desired records only if a value is specified for each field in the search key.

With respect to a composite key index, in a range query not all fields in the search key are bound to constants. For example, we can ask to retrieve all data entries with *age* = 20; this query implies that any value is acceptable for the *sal* field. As another example of a range query, we can ask to retrieve all data entries with *age* < 30 and *sal> 40.*

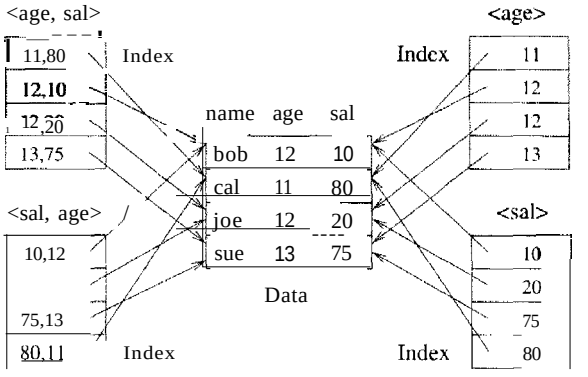


Figure 8.5 Composite Key Indexes

Nate that the index cannot help on the query *sal* > 40, because, intuitively, the index organizes records by *age* first and then *sal.* If *age* is left unspecified, qualifying records could be spread across the entire index. We say that an index **matches** a selection condition if the index can be used to retrieve just the tuples that satisf:y the condition. For selections of the form *condition* $\wedge \ldots \wedge$ *condition,* we can define when an index matches the selection as 1'01lows:[4] For a hash index, a selection matches the index if it includes an equality condition ('field = constant') on every field in the composite search key for the index. For a tree index, a selection matches the index if it includes an equality or range condition on a *prefix* of the composite search key. (As examples, *(age)* and $\langle age, sal, department)$ are prefixes of key *(age, sal,* $department\rangle$, but $\langle age, department)$ and *(sal,* $department\rangle$ are not.)

Trade-offs in Choosing Composite Keys

A composite key index can support a broader range of queries because it matches more selection conditions. Further, since data entries in a composite index contain more information about the data record (i.e., more fields than a single-attribute index), the opportunities for index-only evaluation strategies are increased. (Recall from Section 8.5.2 that an index-only evaluation does not need to access data records, but finds all required field values in the data entries of indexes.)

On the negative side, a composite index must be updated in response to any operation (insert, delete, or update) that modifies *any* field in the search key. A composite index is also likely to be larger than a single-attribute search key

[4] For a more general discussion, see Section 14.2.)

index because the size of entries is larger. For a composite B+ tree index, this also means a potential increase in the number of levels, although key COlnpression can be used to alleviate this problem (see Section 10.8.1).

Design Examples of Composite Keys

Consider the following query, which returns all employees with $20 < age < 30$ and $3000 < sal < 5000$:

```
SELECT  E.eid
FROM    Employees E
WHERE   E.age BETWEEN 20 AND 30
        AND E.sal BETWEEN 3000 AND 5000
```

A composite index on *(age, sal)* could help if the conditions in the WHERE clause are fairly selective. Obviously, a hash index will not help; a B+ tree (or ISAM) index is required. It is also clear that a clustered index is likely to be superior to an unclustered index. For this query, in which the conditions on *age* and *sal* are equally selective, a composite, clustered B+ tree index on *(age, sal)* is as effective as a composite, clustered B+ tree index on *(sal, age).* However, the order of search key attributes can sometimes make a big difference, as the next query illustrates:

```
SELECT  E.eid
FROM    Employees E
WHERE   E.age = 25
        AND E.sal BETWEEN 3000 AND 5000
```

In this query a composite, clustered B+ tree index on *(age, sal)* will give good performance because records are sorted by *age* first and then (if two records have the same *age* value) by *sal.* Thus, all records with *age* = 25 are clustered together. On the other hand, a composite, clustered B+ tree index on *(sal, age)* will not perform as well. In this case, records are sorted by *sal* first, and therefore two records with the same *age* value (in particular, with *age* = 25) may be quite far apart. In effect, this index allows us to use the range selection on *sal,* but not the equality selection on *age,* to retrieve tuples. (Good performance on both variants of the query can be achieved using a single *spatial* index. \:Ye discuss spatial indexes in Chapter 28.)

Composite indexes are also useful in dealing with many aggregate queries. Consider:

```
SELECT  AVG  (E.sal)
```

```
FROM    Employees E
WHERE   E.age = 25
        AND Ksal BETWEEN 3000 AND 5000
```

A composite B+ tree index on *(age, sal)* allows us to answer the query with an index-only scan. A composite B+ tree index on *(sal, age)* also allows us to answer the query with an index-only scan, although more index entries are retrieved in this case than with an index on *(age, sal)*.

Here is a variation of an earlier example:

```
SELECT    Kdno, COUNT(*)
FROM      Employees E
WHERE     E.sal=lO,OOO
GROUP BY  Kdno
```

An index on *dna* alone does not allow us to evaluate this query with an index-only scan, because we need to look at the *sal* field of each tuple to verify that *sal* = 10, 000. However, we can use an index-only plan if we have a composite B+ tree index on *(sal, dna)* or *(dna, sal)*. In an index with key *(sal, dno)*, all data entries with *sal* = 10,000 are arranged contiguously (whether or not the index is clustered). Further, these entries are sorted by *dna*, making it easy to obtain a count for each *dna* group. Note that we need to retrieve only data entries with *sal* = 10, 000.

It is worth observing that this strategy does not work if the WHERE clause is modified to use *sal* > 10, 000. Although it suffices to retrieve only index data entries-that is, an index-only strategy still applies-these entries must now be sorted by *dna* to identify the groups (because, for example, two entries with the same *dna* but different *sal* values may not be contiguous). An index with key *(dna, sal)* is better for this query: Data entries with a given *dna* value are stored together, and each such group of entries is itself sorted by *sal*. For each *dna* group, we can eliminate the entries with *sal* not greater than 10,000 and count the rest. (Using this index is less efficient than an index-only scan with key *(sal, dna)* for the query with *sal* = 10, 000, because we must read all data entries. Thus, the choice between these indexes is influenced by which query is more common.)

As another example, suppose we want to find the minimum *sal* for each *dna:*

```
SELECT    Kdno, MIN(E.sal)
FROM      Employees E
GROUP BY  E.dno
```

An index on *dna* alone does not allow us to evaluate this query with an index-only scan. However, we can use an index-only plan if we have a composite B+ tree index on $\langle dno, sal \rangle$. Note that all data entries in the index with a given *dna* value are stored together (whether or not the index is clustered). Further, this group of entries is itself sorted by *8al.* An index on *(sal, dna)* enables us to avoid retrieving data records, but the index data entries must be sorted on *dno.*

8.5.4 Index Specification in SQL: 1999

A natural question to ask at this point is how we can create indexes using SQL. The SQL:1999 standard does *not* include any statement for creating or dropping index structures. In fact, th.e standard does not even require SQL implementations to support indexes! In practice, of course, every commercial relational DBMS supports one or more kinds of indexes. The following command to create a B+ tree index-we discuss B+ tree indexes in Chapter 10—is illustrative:

```
CREATE INDEX IndAgeRating ON Students
      WITH STRUCTURE = BTREE,
           KEY = (age, gpa)
```

This specifies that a B+ tree index is to be created on the Students table using the concatenation of the *age* and *gpa* columns as the key. Thus, key values are pairs of the form *(age, gpa)*, and there is a distinct entry for each such pair. Once created, the index is automatically maintained by the DBMS adding or removing data entries in response to inserts or deletes of records on the Students relation.

8.6 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- 'Where does a DBMS store persistent data? How does it bring data into main memory for processing? What DBMS component reads and writes data from main memory, and what is the unit of I/O? (Section 8.1)
- 'What is a *file organization?* What is an *index?* What is the relationship between files and indexes? Can we have several indexes on a single file of records? Can an index itself store data records (i.e., act as a file)? (Section 8.2)
- What is the *search key* for an index? What is a *data entry* in an index? (Section 8.2)

- What is a *clustered* index? What is a *primary index?* How many clustered indexes can you build on a file? How many unclustered indexes can you build? **(Section 8.2.1)**
- How is data organized in a hash-based index? \Vhen would you use a hash-based index? **(Section 8.3.1)**
- How is data organized in a tree-based index? When would you use a tree-based index? **(Section 8.3.2)**
- Consider the following operations: *scans, equality and range selections, inserts, and deletes,* and the following file organizations: *heap files, sorted files, clustered files, heap files with an unclustered tree index on the search key,* and *heap files with an unclusteTed hash index.* Which file organization is best suited for each operation? **(Section 8.4)**
- What are the main contributors to the cost of database operations? Discuss a simple cost model that reflects this. **(Section 8.4.1)**
- How does the expected workload influence physical database design decisiems such as what indexes to build? Why is the choice of indexes a central aspect of physical database design? **(Section 8.5)**
- What issues are considered in using clustered indexes? What is an *indcl;-only* evaluation method? What is its primary advantage? **(Section 8.5.2)**
- What is a *composite 8earch key?* What are the pros and cons of composite search keys? **(Section 8.5.3)**
- What SQL commands support index creation? **(Section 8.5.4)**

EXERCISES

Exercise 8.1 Answer the following questions about data on external storage in a DBMS:

1. \Vhy does a DBMS store data on external storage?
2. Why are I/O costs important in a DBMS?
3. What is a record id? Given a record's id, how many I/Os are needed to fetch it into main memory?
4. \Vhat is the role of the buffer manager in a DBMS? What is the role of the disk space manager? How do these layers interact with the file and access methods layer?

Exercise 8.2 Answer the following questions about files and indexes:

1. What operations arc supported by the file of records abstraction?
2. \Vhat is an index on a file of records? \Nhat is a search key for an index? Why do we need indexes?

	name		age	gpa
53831	Madayan	madayan@music	11	1.8
53832	Gulclu	guldu@music	12	2.0
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	19	3.2
53650	Smith	smith@math	19	3.8

Figure 8.6 An Instance of the Students Relation, Sorted by *age*

3. What alternatives are available for the data entries in an index?
4. What is the difference between a primary index and a secondary index? \Vhat is a duplicate data entry in an index? Can a primary index contain duplicates?
5. What is the difference between a clustered index and an unclustered index? If an index contains data records as 'data entries,' can it be unclustered?
6. How many clustered indexes can you create on a file? Woule! you always create at least one clustered index for a file?
7. Consider Alternatives (1), (2) and (3) for 'data entries' in an index, as discussed in Section 8.2 . Are all of them suitable for secondary indexes? Explain.

Exercise 8.3 Consider a relation stored as a randomly ordered file for which the only index is an unclustered index on a field called *sal*. If you want to retrieve all records with $sal > 20$, is using the index always the best alternative? Explain.

Exercise 8.4 Consider the instance of the Students relation shown in Figure 8.6, sorted by *age*: For the purposes of this question, assume that these tuples are stored in a sorted file in the order shown; the first tuple is on page 1 the second tuple is also on page 1; and so on. Each page can store up to three data records; so the fourth tuple is on page 2.

Explain what the data entries in each of the following indexes contain. If the order of entries is significant, say so and explain why. If such all index cannot be constructeel, say so and explain why.

1. An unclustereel index on *age* using Alternative (1).
2. An unclusterecl index on *age* using Alternative (2).
3. An unclustered index on *age* using Alternative (3).
4. A clustered index on *age* using Alternative (1).
5. A clustered index on *age* using Alt.ernative (2).
6. A clustered index on *age* using Alternative (3).
7. An unc:lustered index on *gpa* using Alternative (1).
8. An unclustered index on *gpa* using Alternative (2).
9. An unclustered index on *gpa* using Alternative (3).
10. A clustered index on *gpa* using Alternative (1).
11. A clustered index on *gpa* using Alternative (2).
12. A clustered index on *gpa* using Alternative (3).

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap file					
Sorted file					
Clustered file					
Unclustered tree index					
Unclustered hash index					

Figure 8.7 I/O Cost Comparison

Exercise 8.5 Explain the difference between Hash indexes and B+-tree indexes. In particular, discuss how equality and range searches work, using an example.

Exercise 8.6 Fill in the I/O costs in Figure 8.7.

Exercise 8.7 If you were about to create an index on a relation, what considerations would guide your choice? Discuss:

1. The choice of primary index.
2. Clustered versus unclustered indexes.
3. Hash versus tree indexes.
4. The use of a sorted file rather than a tree-based index.
5. Choice of search key for the index. What is a composite search key, and what considerations are made in choosing composite search keys? What are index-only plans, and what is the influence of potential index-only evaluation plans on the choice of search key for an index?

Exercise 8.8 Consider a delete specified using an equality condition. For each of the five file organizations, what is the cost if no record qualifies? What is the cost if the condition is not on a key?

Exercise 8.9 What main conclusions can you draw from the discussion of the five basic file organizations discussed in Section 8.4? Which of the five organizations would you choose for a file where the most frequent operations are as follows?

1. Search for records based on a range of field values.
2. Perform inserts and scans, where the order of records docs not matter.
3. Search for a record based on a particular field value.

Exercise 8.10 Consider the following relation:

Emp(*eid:* integer, *sal:* integer, *age:* real, *did:* integer)

There is a clustered index on *cid* and an llnclustered index on *age*.

1. How would you use the indexes to enforce the constraint that *eid* is a key?
2. Give an example of an update that is *definitely speeded up* because of the available indexes. (English description is sufficient.)

3. Give an example of an update that is *definitely slowed down* because of the indexes. (English description is sufficient.)
4. Can you give an example of an update that is neither speeded up nor slowed down by the indexes?

Exercise 8.11 Consider the following relations:

Emp(*eid:* integer, *ename:* varchar, *sal:* integer, *age:* integer, *did:* integer)
Dept(did: integer, *budget:* integer, *floor:* integer, *mgr_eid:* integer)

Salaries range from $10,000 to $100,000, ages vary from 20 to 80, each department has about five employees on average, there are 10 floors, and budgets vary from $10,000 to $1 million. You can assume uniform distributions of values.

For each of the following queries, which of the listed index choices would you choose to speed up the query? If your database system does not consider index-only plans (i.e., data records are always retrieved even if enough information is available in the index entry), how would your answer change? Explain briefly.

1. Query: *Print ename, age, and sal for all employees.*
   (a) Clustered hash index on *(ename, age, sal)* fields of Emp.
   (b) Unclustered hash index on *(ename, age, sal)* fields of Emp.
   (c) Clustered B+ tree index on *(ename, age, sal)* fields of Emp.
   (d) Unclustered hash index on *(eid, did)* fields of Emp.
   (e) No index.
2. Query: *Find the dids of departments that are on the 10th floor and have a budget of less than $15,000.*
   (a) Clustered hash index on the *floor* field of Dept.
   (b) Unclustered hash index on the *floor'* field of Dept.
   (c) Clustered B+ tree index on *(floor, budget)* fields of Dept.
   (d) Clustered B+ tree index on the *budget* field of Dept.
   (e) No index.

PROJECT-BASED EXERCISES

Exercise 8.12 Answer the following questions:

1. What indexing techniques are supported in Minibase?
2. What alternatives for data entries are supported'?
3. Are clustered indexes supported?

BIBLIOGRAPHIC NOTES

Several books discuss file organization in detail [29, 312, 442, 531, 648, 695, 775].

Bibliographic: notes for hash-indexes and B+-trees are included in Chapters 10 and 11.

9

STORING DATA: DISKS AND FILES

- What are the different kinds of memory in a computer system?
- What are the physical characteristics of disks and tapes, and how do they affect the design of database systems?
- What are RAID storage systems, and what are their advantages?
- How does a DBMS keep track of space on disks? How does a DBMS access and modify data on disks? What is the significance of *pages* as a unit of storage and transfer?
- How does a DBMS create and maintain files of records? How are records arranged on pages, and how are pages organized within a file?
- Key concepts: memory hierarchy, persistent storage, random versus sequential devices; physical disk architecture, disk characteristics, seek time, rotational delay, transfer time; RAID, striping, mirroring, RAID levels; disk space manager; buffer manager, buffer pool, replacement policy, prefetching, forcing; file implementation, page organization, record organization

A memory is what is left when something happens and does not cornpletely unhappen.

. Edward DeBono

This chapter initiates a study of the internals of an RDBivIS. In terms of the DBMS architecture presented in Section 1.8, it covers the disk space manager,

the buffer manager, and implementation-oriented aspects of the *Jiles and access methods* layer.

Section 9.1 introduces disks and tapes. Section 9.2 describes RAID disk systems. Section 9.3 discusses how a DBMS manages disk space, and Section 9.4 explains how a DBMS fetches data from disk into main memory. Section 9.5 discusses how a collection of pages is organized into a file and how auxiliary data structures can be built to speed up retrieval of records from a file. Section 9.6 covers different ways to arrange a collection of records on a page, and Section 9.7 covers alternative formats for storing individual records.

9.1 THE MEMORY HIERARCHY

Memory in a computer system is arranged in a hierarchy, as shown in Figure 9.1. At the top, we have primary storage, which consists of cache and main memory and provides very fast access to data. Then comes secondary storage, which consists of slower devices, such as magnetic disks. Tertiary storage is the slowest class of storage devices; for example, optical disks and tapes. Currently, the cost of a given amount of main memory is about 100 times

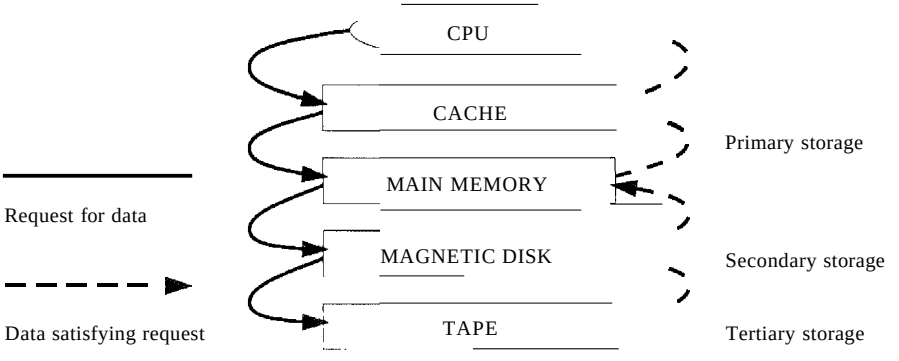


Figure 9.1 The Ivlemory Hierarchy

the cost of the same amount of disk space, and tapes are even less expensive than disks. Slower storage devices such as tapes and disks play an important role in database systems because the amount of data is typically very large. Since buying enough main memory to store all data is prohibitively expensive, we must store data on tapes and disks and build database systems that can retrieve data from lower levels of the memory hierarchy into main mernory as needed for processing.

There are reasons other than cost for storing data on secondary and tertiary storage. On systems with 32-bit addressing, only 2^{32} bytes can be directly referenced in main memory; the number of data objects may exceed this number! Further, data must be maintained across program executions. This requires storage devices that retain information when the computer is restarted (after a shutdown or a crash); we call such storage nonvolatile. Primary storage is usually volatile (although it is possible to make it nonvolatile by adding a battery backup feature), whereas secondary and tertiary storage are nonvolatile.

Tapes are relatively inexpensive and can store very large amounts of data. They are a good choice for *archival* storage, that is, when we need to maintain data for a long period but do not expect to access it very often. A Quantum DLT 4000 drive is a typical tape device; it stores 20 GB of data and can store about twice as much by compressing the data. It records data on 128 *tape tracks*, which can be thought of as a linear sequence of adjacent bytes, and supports a sustained transfer rate of 1.5 MB/sec with uncompressed data (typically 3.0 MB/sec with compressed data). A single DLT 4000 tape drive can be used to access up to seven tapes in a stacked configuration, for a maximum compressed data capacity of about 280 GB.

The main drawback of tapes is that they are sequential access devices. We must essentially step through all the data in order and cannot directly access a given location on tape. For example, to access the last byte on a tape, we would have to wind through the entire tape first. This makes tapes unsuitable for storing *operational data*, or data that is frequently accessed. Tapes are mostly used to back up operational data periodically.

9.1.1 Magnetic Disks

Magnetic disks support direct access to a desired location and are widely used for database applications. A DBMS provides seamless access to data on disk; applications need not worry about whether data is in main memory or disk. To understand how disks work, eonsider Figure 9.2, which shows the structure of a disk in simplified form.

Data is stored on disk in units called disk blocks. A disk block is a contiguous sequence of bytes and is the unit in which data is written to a disk and read from a disk. Bloc:ks are arranged in concentric rings called tracks, on one or more platters. Tracks can be recorded on one or both surfaces of a platter; we refer to platters as single-sided or double-sided, accordingly. The set of all tracks with the same diameter is called a cylinder, because the space occupied by these tracks is shaped like a cylinder; a cylinder contains one track per platter surface. Each track is divided into arcs, called sectors, whose size is a

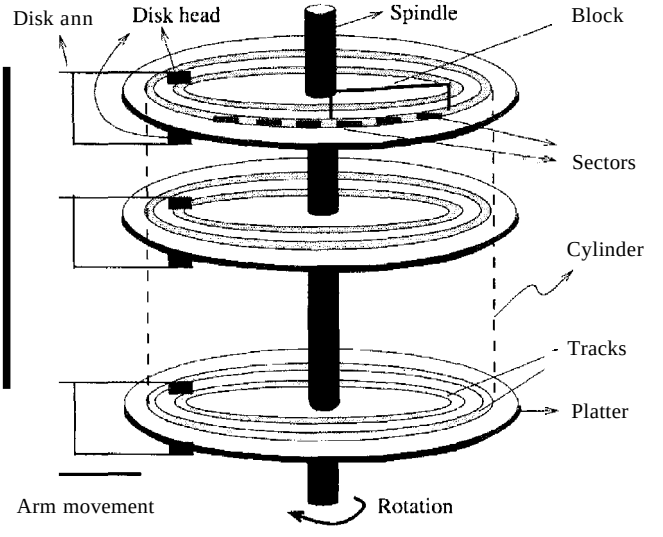


Figure 9.2 Structure of a Disk

characteristic of the disk and cannot be changed. The size of a disk block can be set when the disk is initialized as a multiple of the sector size.

An array of disk heads, one per recorded surface, is moved as a unit; when one head is positioned over a block, the other heads are in identical positions with respect to their platters. To read or write a block, a disk head must be positioned on top of the block.

Current systems typically allow at most one disk head to read or write at any one time. All the disk heads cannot read or write in parallel—this technique would increase data transfer rates by a factor equal to the number of disk heads and considerably speed up sequential scans. The reason they cannot is that it is very difficult to ensure that all the heads are perfectly aligned on the corresponding tracks. Current approaches are both expensive and more prone to faults than disks with a single active heacl. In practice, very few commercial products support this capability and then only in a limited way; for example, two disk heads may be able to operate in parallel.

A disk controller interfaces a disk drive to the computer. It implements commands to read or write a sector by moving the arm assembly and transferring data to and from the disk surfaces. A checksum is computed for when data is written to a sector and stored with the sector. The checksum is computed again when the data on the sector is read back. If the sector is corrupted or the

An Example of a Current Disk: The IBM Deskstar 14GPX. The IBM Deskstar 14GPX is a 3.5 inch, 14.4 GB hard disk with an average seek time of 9.1 milliseconds (msec) and an average rotational delay of 4.17 msec. However, the time to seek from one track to the next is just 2.2 msec, the maximum seek time is 15.5 msec. The disk has five double-sided platters that spin at 7200 rotations per minute. Each platter holds 3.35 GB of data, with a density of 2.6 gigabit per square inch. The data transfer rate is about 13 MB per second. To put these numbers in perspective, observe that a disk access takes about 10 msecs, whereas accessing a main memory location typically takes less than 60 nanoseconds!

read is faulty for some reason, it is very unlikely that the checksum computed when the sector is read matches the checksum computed when the sector was written. The controller computes checksums, and if it detects an error, it tries to read the sector again. (Of course, it signals a failure if the sector is corrupted and read fails repeatedly.)

While direct access to any desired location in main memory takes approximately the same time, determining the time to access a location on disk is more complicated. The time to access a disk block has several components. **Seek time** is the time taken to move the disk heads to the track on which a desired block is located. As the size of a platter decreases, seek times also decrease, since we have to move a disk head a shorter distance. Typical platter diameters are 3.5 inches and 5.25 inches. **Rotational delay** is the waiting time for the desired block to rotate under the disk head; it is the time required for half a rotation all average and is usually less than seek time. **Transfer time** is the time to actually read or write the data in the block once the head is positioned, that is, the time for the disk to rotate over the block.

9.1.2 Performance Implications of Disk Structure

1. Data must be in mernory for the DBMS to operate on it.
2. The unit for data transfer between disk and main memory is a block; if a single item on a block is needed, the entire block is transferred. Reading or writing a disk block is called an I/O (for input/output) operation.
3. The time to read or write a block varies, depending on the location of the data:

$$access\ time = seek\ time + rotational\ delay + transfer\ time$$

These observations imply that the time taken for database operations is affected significantly by how data is stored on disks. The time for moving blocks to

or from disk usually dOlninates the time taken for database operations. To minimize this time, it is necessary to locate data records strategically on disk because of the geometry and mechanics of disks. **In** essence, if two records are frequently used together, we should place them close together. The 'closest' that two records can be on a disk is to be on the same block. **In** decreasing order of closeness, they could be on the same track, the same cylinder, or an adjacent cylinder.

Two records on the same block are obviously as close together as possible, because they are read or written as part of the same block. As the platter spins, other blocks on the track being read or written rotate under the active head. In current disk designs, all the data on a track can be read or written in one revolution. After a track is read or written, another disk head becomes active, and another track in the same cylinder is read or written. This process continues until all tracks in the current cylinder are read or written, and then the arm assembly moves (in or out) to an adjacent cylinder. Thus, we have a natural notion of 'closeness' for blocks, which we can extend to a notion of *next* and *previous* blocks.

Exploiting this notion of next by arranging records so they are read or written sequentially is very important in reducing the time spent in disk l/Os. Sequential access minimizes seek time and rotational delay and is much faster than random access. (This observation is reinforced and elaborated in Exercises 9.5 and 9.6, and the reader is urged to work through them.)

9.2 REDUNDANT ARRAYS OF INDEPENDENT DISKS

Disks are potential bottlenecks for system performance and storage system reliability. Even though disk performance has been improving continuously, microprocessor performance has advanced much more rapidly. The performance of microprocessors has improved at about 50 percent or more per year, but disk access times have improved at a rate of about 10 percent per year and disk transfer rates at a rate of about 20 percent per year. **In** addition, since disks contain mechanical elements, they have much higher failure rates than electronic parts of a computer system. **If** a disk fails, all the data stored on it is lost.

A **disk array** is an arrangement of several disks, organized to increase performance and improve reliability of the resulting storage system. Performance is increased through data striping. Data striping distributes data over several disks to give the impression of having a single large, very fast disk. Reliability is improved through **redundancy.** Instead of having a single copy of the data, redundant information is maintained. The redundant information is care-

fully organized so that, in case of a disk failure, it can be used to reconstruct the contents of the failed disk. Disk arrays that implement a combination of data striping and redundancy are called redundant arrays of independent disks, or in short, **RAID**.[1] Several RAID organizations, referred to as RAID levels, have been proposed. Each RAID level represents a different trade-off between reliability and performance.

In the remainder of this section, we first discuss data striping and redundancy and then introduce the RAID levels that have become industry standards.

9.2.1 Data Striping

A disk array gives the user the abstraction of having a single, very large disk. If the user issues an I/O request, we first identify the set of physical disk blocks that store the data requested. These disk blocks may reside on a single disk in the array or may be distributed over several disks in the array. Then the set of blocks is retrieved from the disk(s) involved. Thus, how we distribute the data over the disks in the array influences how many disks are involved when an I/O request is processed.

In data striping, the data is segmented into equal-size partitions distributed over multiple disks. The size of the partition is called the striping unit. The partitions are usually distributed using a round-robin algorithm: If the disk array consists of D disks, then partition i is written onto disk i mod D.

As an example, consider a striping unit of one bit. Since any D successive data bits are spread over all D data disks in the array, all I/O requests involve all disks in the array. Since the smallest unit of transfer from a disk is a block, each I/O request involves transfer of at least D blocks. Since we can read the D blocks from the D disks in parallel, the transfer rate of each request is D times the transfer rate of a single disk; each request uses the aggregated bandwidth of all disks in the array. But the disk access time of the array is basically the access time of a single disk, since all disk heads have to move for all requests. Therefore, for a disk array with a striping unit of a single bit, the number of requests per time unit that the array can process and the average response time for each individual request are similar to that of a single disk.

As another exarhple, consider a striping unit of a disk block. In this case, I/O requests of the size of a disk block are processed by one disk in the array. If many I/O requests of the size of a disk block are made, and the requested

[1] Historically, the *J* in RAID stood for inexpensive, as a large number of small disks was much more economical than a single very large disk. Today, such very large disks are not even manufactured—a sign of the impact of RAID.

Redundancy Schemes: Alternatives to the parity scheme include schemes based on Hamming codes and Reed-Solomon codes. In addition to recovery from single disk failures, Hamming codes can identify which disk failed. Reed-Solomon codes can recover from up to two simultaneous disk failures. A detailed discussion of these schemes is beyond the scope of our discussion here; the bibliography provides pointers for the interested reader.

blocks reside on different disks, we can process all requests in parallel and thus reduce the average response time of an I/O request. Since we distributed the striping partitions round-robin, large requests of the size of many contiguous blocks involve all disks. We can process the request by all disks in parallel and thus increase the transfer rate to the aggregated bandwidth of all D disks.

9.2.2 Redundancy

While having more disks increases storage system performance, it also lowers overall storage system reliability. Assume that the mean-time-to-failure (MTTF), of a single disk is 50,000 hours (about 5.7 years). Then, the MTTF of an array of 100 disks is only 50,000/100 = 500 hours or about 21 days, assuming that failures occur independently and the failure probability of a disk does not change over time. (Actually, disks have a higher failure probability early and late in their lifetimes. Early failures are often due to undetected manufacturing defects; late failures occur since the disk wears out. Failures do not occur independently either: consider a fire in the building, an earthquake, or purchase of a set of disks that come from a 'bad' manufacturing batch.)

Reliability of a disk array can be increased by storing redundant information. If a disk fails, the redundant information is used to reconstruct the data on the failed disk. Redundancy can immensely increase the MTTF of a disk array. When incorporating redundancy into a disk array design, we have to make two choices. First, we have to decide where to store the redundant information. We can either store the redundant information on a small number of check disks or distribute the redundant information uniformly over all disks.

The second choice we have to make is how to compute the redundant information. Most disk arrays store parity information: In the parity scheme, an extra check disk contains information that can be used to recover from failure of anyone disk in the array. Assume that we have a disk array with D disks and consider the first bit on each data disk. Suppose that i of the D data bits are 1. The first bit on the check disk is set to 1 if i is odd; otherwise, it is set to

0. This bit on the check disk is called the parity of the data bits. The check disk contains parity information for each set of corresponding D data bits.

To recover the value of the first bit of a failed disk we first count the number of bits that are 1 on the D - 1 nonfailed disks; let this number be *j*. If *j* is odd and the parity bit is 1, or if *j* is even and the parity bit is 0, then the value of the bit on the failed disk must have been 0. Otherwise, the value of the bit on the failed disk must have been 1. Thus, with parity we can recover from failure of anyone disk. Reconstruction of the lost information involves reading all data disks and the check disk.

For example, with an additional 10 disks with redundant information, the MTTF of our example storage system with 100 data disks can be increased to more than 250 years! "What is more important, a large MTTF implies a small failure probability during the actual usage time of the storage system, which is usually much smaller than the reported lifetime or the MTTF. (Who actually uses lO-year-old disks?)

In a RAID system, the disk array is partitioned into reliability groups, where a reliability group consists of a set of *data disks* and a set of *check disks.* A common *7'cdundancy scheme* (see box) is applied to each group. The number of check disks depends on the RAID level chosen. In the remainder of this section, we assume for ease of explanation that there is only one reliability group. The reader should keep in mind that actual RAID implementations consist of several reliability groups, and the number of groups plays a role in the overall reliability of the resulting storage system.

9.2.3 Levels of Redundancy

Throughout the discussion of the different RAID levels, we consider sample data that would just fit on four disks. That is, with no RAID technology our storage system would consist of exactly four data disks. Depending on the RAID level chosen, the number of additional disks varies from zero to four.

Level 0: Nonredundant

A RAID Level 0 system uses data striping to increase the maximum bandwidth available. No redundant information is maintained. While being the solution with the lowest cost, reliability is a problem, since the MTTF decreases linearly with the number of disk drives in the array. RAID Level 0 has the best write performance of all RAID levels, because absence of redundant information implies that no redundant information needs to be updated! Interestingly, RAID Level 0 docs not have the best read perfonnancc of all RAID levels, since sys-

tems with redundancy have a choice of scheduling disk accesses, as explained in the next section.

In our example, the RAID Level a solution consists of only four data disks. Independent of the number of data disks, the effective space utilization for a RAID Level a system is always 100 percent.

Levell: Mirrored

A RAID Level 1 system is the most expensive solution. Instead of having one copy of the data, two identical copies of the data on two different disks are lnaintained. This type of redundancy is often called mirroring. Every write of a disk block involves a write on both disks. These writes may not be performed simultaneously, since a global system failure (e.g., due to a power outage) could occur while writing the blocks and then leave both copies in an inconsistent state. Therefore, we always write a block on one disk first and then write the other copy on the mirror disk. Since two copies of each block exist on different disks, we can distribute reads between the two disks and allow *parallel reads* of different disk blocks that conceptually reside on the same disk. A read of a block can be scheduled to the disk that has the smaller expected access time. RAID Level 1 does not stripe the data over different disks, so the transfer rate for a single request is comparable to the transfer rate of a single disk.

In our example, we need four data and four check disks with mirrored data for a RAID Levell implementation. The effective space utilization is 50 percent, independent of the number of data disks.

Level 0+1: Striping and Mirroring

RAID Level 0+l---sometimes also referred to as *RAID Level 10*--combines striping and mirroring. As in RAID Level 1. read requests of the size of a disk block can be scheduled both to a disk and its mirror image. In addition, read requests of the size of several contiguous blocks benefit froll1 the aggregated bandwidth of all disks. Thc cost for writes is analogous to RAID LevelL

As in RAID Level 1, our example with four data disks requires four check disks and the effective space utilization is always 50 percent.

Level 2: **Error-Correcting** Codes

In RAID Level 2, the striping unit is a single bit. The redundancy scheme used is Hamming code. In our example with four data disks, only three check disks

are needed. In general, the number of check disks grows logarithmically with the number of data disks.

Striping at the bit level has the implication that in a disk array with D data disks, the smallest unit of transfer for a read is a set of D blocks. Therefore, Level 2 is good for workloads with many large requests, since for each request, the aggregated bandwidth of all data disks is used. But RAID Level 2 is bad for small requests of the size of an individual block for the same reason. (See the example in Section 9.2.1.) A write of a block involves reading D blocks into main memory, modifying $D + C$ blocks, and writing $D + C$ blocks to disk, where C is the number of check disks. This sequence of steps is called a *read-modify-write* cycle.

For a RAID Level 2 implementation with four data disks, three check disks are needed. In our example, the effective space utilization is about 57 percent. The effective space utilization increases with the number of data disks. For example, in a setup with 10 data disks, four check disks are needed and the effective space utilization is 71 percent. In a setup with 25 data disks, five check disks are required and the effective space utilization grows to 83 percent.

Level 3: **Bit-Interleaved Parity**

While the redundancy schema used in RAID Level 2 improves in terms of cost over RAID Level 1, it keeps more redundant information than is necessary. Hamming code, as used in RAID Level 2, has the advantage of being able to identify which disk has failed. But disk controllers can easily detect which disk has failed. Therefore, the check disks do not need to contain information to identify the failed disk. Information to recover the lost data is sufficient. Instead of using several disks to store Hamming code, RAID Level 3 has a single check disk with parity information. Thus, the reliability overhead for RAID Level 3 is a single disk, the lowest overhead possible.

The performance characteristics of RAID Levels 2 and 3 are very similar. RAID Level 3 can also process only one I/O at a time, the minimum transfer unit is D blocks, and a write requires a read-modify-write cycle.

Level 4: **Block-Interleaved Parity**

RAID Level 4 has a striping unit of a disk block, instead of a single bit as in RAID Level 3. Block-level striping has the advantage that read requests of the size of a disk block can be served entirely by the disk where the requested block resides. Large read requests of several disk blocks can still utilize the aggregated bandwidth of the D disks.

The \vrite of a single block still requires a read-modify-write cycle, but only one data disk and the check disk are involved. The parity on the check disk can be updated without reading all D disk blocks, because the new parity can be obtained by noticing the differences between the old data block and the new data block and then applying the difference to the parity block on the check disk:

NewParity = (OldData XOR NewData) XOR OldParity

The read-modify-write cycle involves reading of the old data block and the old parity block, modifying the two blocks, and writing them back to disk, resulting in four disk accesses per write. Since the check disk is involved in each write, it can easily become the bottleneck.

RAID Level 3 and 4 configurations with four data disks require just a single check disk. In our example, the effective space utilization is 80 percent. The effective space utilization increases with the number of data disks, since always only one check disk is necessary.

Level 5: **Block-Interleaved Distributed Parity**

RAID Level 5 improves on Level 4 by distributing the parity blocks uniformly over all disks, instead of storing them on a single check disk. This distribution has two advantages. First, several write requests could be processed in parallel, since the bottleneck of a unique check disk has been eliminated. Second, read requests have a higher level of parallelism. Since the data is distributed over all disks, read requests involve all disks, whereas in systems with a dedicated check disk the check disk never participates in reads.

A RAID Level 5 system has the best performance of all RAID levels with redundancy for small and large read ancllarge write requests. Small writes still require a read-modify-write cycle and are thus less efficient than in RAID Level 1.

In our example, the corresponding RAID Level 5 system has five disks overall and thus the effective spa,ce utilization is the same as in RAID Levels 3 and 4.

Level 6: P+Q Redundancy

The motivation for RAID Level 6 is the observation that recovery from failure of a single disk is not sufficient in very large disk arrays. First, in large disk arrays, a second disk lllight fail before replacement of an already failed disk

could take place. In addition, the probability of a disk failure during recovery of a failed disk is not negligible.

A RAID Level 6 system uses Reed-Solomon codes to be able to recover from up to two simultaneous disk failures. RAID Level 6 requires (conceptually) two check disks, but it also uniformly distributes redundant information at the block level as in RAID Level 5. Thus. the performance characteristics for small and large read requests and for large write requests are analogous to RAID Level 5. For small writes, the read-modify-write procedure involves six instead of four disks as compared to RAID Level 5, since two blocks with redundant information need to be updated.

For a RAID Level 6 system with storage capacity equal to four data disks, six disks are required. In our example, the effective space utilization is 66 percent.

9.2.4 Choice of RAID Levels

If data loss is not an issue, RAID Level 0 improves overall system performance at the lowest cost. RAID Level 0+1 is superior to RAID Level 1. The main application areas for RAID Level 0+1 systems are small storage subsystems where the cost of mirroring is moderate. Sometimes, RAID Level 0+1 is used for applications that have a high percentage of writes in their workload, since RAID Level 0+1 provides the best write performance. RAID Levels 2 and 4 are always inferior to RAID Levels 3 and 5, respectively. RAID Level 3 is appropriate for workloads consisting mainly of large transfer requests of several contiguous blocks. The performance of a RAID Level 3 system is bad for workloads with many small requests of a single disk block. RAID Level 5 is a good general-purpose solution. It provides high performance for large as well as small requests. RAID Level 6 is appropriate if a higher level of reliability is required.

9.3 DISK SPACE MANAGEMENT

The lowest level of software in the DBMS architecture discussed in Section 1.8, called the disk space manager, manages space on disk. Abstractly, the disk space manager supports the concept of a page as a unit of data and provides commands to allocate or deallocate a page and read or write a page. The size of a page is chosen to be the size of a disk block and pages are stored as disk blocks so that reading or writing a page can be done in one disk I/O.

It is often useful to allocate a sequence of pages as a *contiguous* sequence of blocks to hold data frequently accessed in sequential order. This capability is essential for exploiting the advantages of sequentially accessing disk blocks,

which we discussed earlier in this chapter. Such a capability, if desired, must be provided by the disk space manager to higher-level layers of the DBMS.

The disk space manager hides details of the underlying hardware (and possibly the operating system) and allows higher levels of the software to think of the data as a collection of pages.

9.3.1 Keeping Track of Free Blocks

A database grows and shrinks as records are inserted and deleted over time. The disk space manager keeps track of which disk blocks are in usc, in addition to keeping track of which pages are on which disk blocks. Although it is likely that blocks are initially allocated sequentially on disk, subsequent allocations and deallocations could in general create 'holes.'

One way to keep track of block usage is to maintain a list of free blocks. As blocks are deallocated (by the higher-level software that requests and uses these blocks), we can add them to the free list for future use. A pointer to the first block on the free block list is stored in a known location on disk.

A second way is to maintain a bitmap with one bit for each disk block, which indicates whether a block is in use or not. A bitmap also allows very fast identification and allocation of contiguous areas on disk. This is difficult to accomplish with a linked list approach.

9.3.2 Using OS File Systems to Manage Disk Space

Operating systems also manage space on disk. Typically, an operating system supports the abstraction of a *file as a sequence of bytes.* The as manages space on the disk and translates requests, such as "Read byte i of file *f*," into corresponding low-level instructions: "Read block m of track *t* of cylinder c of disk *d.*" A database disk space manager could he built using OS files. For example, the entire database could reside in one or more as files for which a number of blocks are allocated (by the aS) and initialized. The disk space manager is then responsible for managing the space in these OS files.

Many database systems do not rely on the as file system and instead do their own disk management, either from scratch or by extending as facilities. The reasons are practical as well as technical One practical reason is that a DBMS vendor who wishes to support several as platfonns cannot assume features specific to any OS, for portability, and would therefore try to make the DBMS code as self-contained as possible. A technical reason is that on a :32-bit systern, the la.rgest file size is 4 GB. whereas a DBMS may want to access a single file

larger than that. A related problem is that typical as files cannot span disk devices, which is often desirable or even necessary in a DBMS. Additional technical reasons why a DBMS does not rely on the as file system are outlined in Section 9.4.2.

9.4 BUFFER MANAGER

To understand the role of the buffer manager, consider a simple example. Suppose that the database contains 1 million pages, but only 1000 pages of main memory are available for holding data. Consider a query that requires a scan of the entire file. Because all the data cannot be brought into main memory at one time, the DBMS must bring pages into main memory as they are needed and, in the process, decide what existing page in main memory to replace to make space for the new page. The policy used to decide which page to replace is called the replacement policy.

In terms of the DBMS architecture presented in Section 1.8, the buffer manager is the software layer responsible for bringing pages from disk to main memory as needed. The buffer manager manages the available main memory by partitioning it into a collection of pages, which we collectively refer to as the buffer pool. The main memory pages in the buffer pool are called frames; it is convenient to think of them as slots that can hold a page (which usually resides on disk or other secondary storage media).

Higher levels of the DBMS code can be written without worrying about whether data pages are in memory or not; they ask the buffer manager for the page, and it is brought into a frame in the buffer pool if it is not already there. Of course, the higher-level code that requests a page must also release the page when it is no longer needed, by informing the buffer manager, so that the frame containing the page can be reused. The higher-level code must also inform the buffer manager if it modifies the requested page; the buffer manager then makes sure that the change is propagated to the copy of the page on disk. Buffer management is illustrated in Figure 9.3.

In addition to the buffer pool itself, the buffer manager maintains some bookkeeping information and two variables for each frame in the pool: *pirLcount* and *dirty*. The number of times that the page currently in a given frame has been requested but not released—the number of current users of the page--is recorded in the *pin_count* variable for that frame. The Boolean variable *dirty* indicates whether the page has been modified since it was brought into the buffer pool from disk.

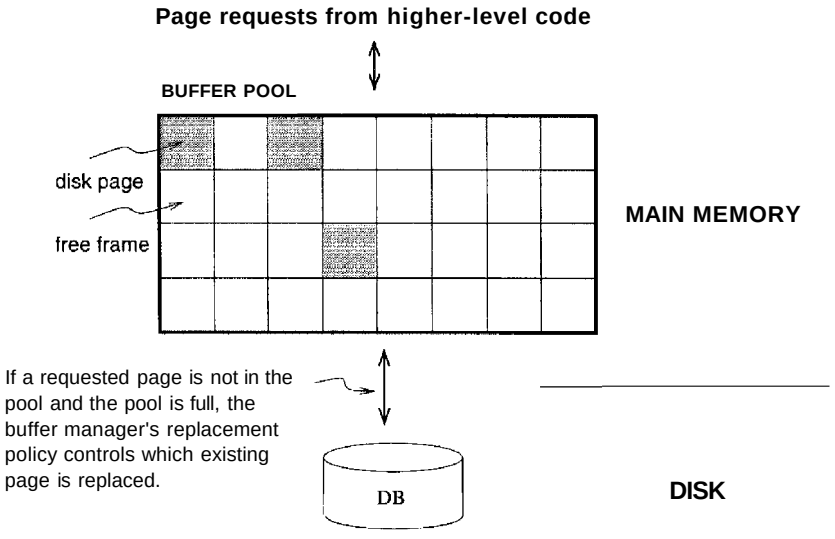


Figure 9.3 The Buffer Pool

Initially, the *pin_count* for every frame is set to 0, and the *dirty* bits are turned off. When a page is requested the buffer manager does the following:

1. Checks the buffer pool to see if some frame contains the requested page and, if so, increments the *pin_count* of that frame. If the page is not in the pool, the buffer manager brings it in as follows:
   (a) Chooses a frame for replacement, using the replacement policy, and increments its *pin_count.*
   (b) If the *dirty* bit for the replacement frame is on, writes the page it contains to disk (that is, the disk copy of the page is overwritten with the contents of the frame).
   (c) Reads the requested page into the replacement frame.
2. Returns the (main memory) address of the frame containing the requested page to the requestor.

Incrementing *pin_count* is often called **pinning** the requested page in its frame. When the code that calls the buffer manager and requests the page subsequently calls the buffer manager and releases the page, the *pin_count* of the frame containing the requested page is decremented. This is called **unpinning** the page. If the requestor has modified the page, it also informs the buffer manager of this at the time that it unpins the page, and the *dirty* bit for the frame is set.

The buffer manager will not read another page into a frame until its *pin_count* becomes 0, that is, until all requestors of the page have unpinned it.

If a requested page is not in the buffer pool and a free frame is not available in the buffer pool, a frame with *pin_count* 0 is chosen for replacement. If there are many such frames, a frame is chosen according to the buffer manager's replacement policy. We discuss various replacement policies in Section 9.4.1.

When a page is eventually chosen for replacement, if the *dirty* bit is not set, it means that the page has not been modified since being brought into main memory. Hence, there is no need to write the page back to disk; the copy on disk is identical to the copy in the frame, and the frame can simply be overwritten by the newly requested page. Otherwise, the modifications to the page must be propagated to the copy on disk. (The crash recovery protocol may impose further restrictions, as we saw in Section 1.7. For example, in the Write-Ahead Log (WAL) protocol, special log records are used to describe the changes made to a page. The log records pertaining to the page to be replaced may well be in the buffer; if so, the protocol requires that they be written to disk *before* the page is written to disk.)

If no page in the buffer pool has *pin_count* 0 and a page that is not in the pool is requested, the buffer manager must wait until some page is released before responding to the page request. In practice, the transaction requesting the page may simply be aborted in this situation! So pages should be released—by the code that calls the buffer manager to request the page- as soon as possible.

A good question to ask at this point is, "What if a page is requested by several different transactions?" That is, what if the page is requested by programs executing independently on behalf of different users? Such programs could make conflicting changes to the page. The locking protocol (enforced by higher-level DBMS code, in particular the transaction manager) ensures that each transaction obtains a shared or exclusive lock before requesting a page to read or rnodify. Two different transactions cannot hold an exclusive lock on the same page at the same time; this is how conflicting changes are prevented. The buffer rnanager simply assumes tha.t the appropriate lock has been obtained before a page is requested.

9.4.1 Buffer Replacement Policies

The policy used to choose an unpinned page for replacement can affect the time taken for database operations considerably. Of the man,Y alternative policies, each is suitable in different situations.

The best-known replacement policy is least recently used (LRU). This can be implemented in the buffer manager using a queue of pointers to frames with *pin_count* 0. A frame is added to the end of the queue when it becomes a candidate for replacement (that is, when the *pin_count* goes to 0). The page chosen for replacement is the one in the frame at the head of the queue.

A variant of LRU, called clock replacement, has similar behavior but less overhead. The idea is to choose a page for replacement using a *current* variable that takes on values 1 through *N*, where *N* is the number of buffer frames, in circular order. We can think of the frames being arranged in a circle, like a clock's face, and *current* as a clock hand moving across the face. To approximate LRU behavior, each frame also has an associated *referenced* bit, which is turned on when the page *pin_count* goes to 0.

The *current* frame is considered for replacement. If the frame is not chosen for replacement, *current* is incremented and the next frame is considered; this process is repeated until some frame is chosen. If the *current* frame has *pin_count* greater than 0, then it is not a candidate for replacement and *current* is incremented. If the *current* frame has the *referenced* bit turned on, the clock algorithm turns the *referenced* bit off and increments *current*—this way, a recently referenced page is less likely to be replaced. If the *current* frame has *pin_count* 0 and its *referenced* bit is off, then the page in it is chosen for replacement. If all frames are pinned in some sweep of the clock hand (that is, the value of *current* is incremented until it repeats), this means that no page in the buffer pool is a replacement candidate.

The LRU and clock policies are not always the best replacement strategies for a database system, particularly if many user requests require sequential scans of the data. Consider the following illustrative situation. Suppose the buffer pool has 10 frames, and the file to be scanned has 10 or fewer pages. Assuming, for simplicity, that there are no competing requests for pages, only the first scan of the file does any I/O. Page requests in subsequent scans always find the desired page in the buffer pool. On the other hand, suppose that the file to be scanned has 11 pages (which is one more than the number of available pages in the buffer pool). Using LRU, every scan of the file will result in reading every page of the file! In this situation, called sequential flooding, LRU is the *worst* possible replacement strategy.

Other replacement policies include first in first out (FIFO) and most recently used (MRU), which also entail overhead similar to LRU, and random, arnong others. The details of these policies should be evident from their names and the preceding discussion of LRU and clock.

Buffer Management in Practice: IBM DB2 and Sybase ASE allow buffers to be partitioned into named pools. Each database, table, or index can be bound to one of these pools. Each pool can be configured to use either LRU or clock replacement in ASE; DB2 uses a variant of clock replacement, with the initial clock value based on the nature of the page (e.g., index non-leaves get a higher starting clock value, which delays their replacement). Interestingly, a buffer pool client in DB2 can explicitly indicate that it *hates* a page, making the page the next choice for replacement. As a special case, DB2 applies MRU for the pages fetched in some utility operations (e.g., RUNSTATS), and DB2 V6 also supports FIFO. Informix and Oracle 7 both maintain a single global buffer pool using LRU; Microsoft SQL Server has a single pool using clock replacement. In Oracle 8, tables can be bound to one of two pools; one has high priority, and the system attempts to keep pages in this pool in memory.

Beyond setting a maximum number of pins for a given transaction, there are typically no features for controlling buffer pool usage on a per-transaction basis. Microsoft SQL Server, however, supports a reservation of buffer pages by queries that require large amounts of memory (e.g., queries involving sorting or hashing).

9.4.2 Buffer Management in DBMS versus OS

Obvious similarities exist between virtual memory in operating systems and buffer management in database management systems. In both cases, the goal is to provide access to more data than will fit in main memory, and the basic idea is to bring in pages from disk to main memory as needed, replacing pages no longer needed in main memory. Why can't we build a DBMS using the virtual memory capability of an OS? A DBMS can often predict the order in which pages will be accessed, or page reference patterns, much more accurately than is typical in an as environment, and it is desirable to utilize this property. Further, a DBMS needs more control over when a page is written to disk than an as typically provides.

A DBMS can often predict reference patterns because most page references are generated by higher-level operations (such as sequential scans or particular implementations of various relational algebra opera.tors) with a known pattern of page accesses. This ability to predict reference patterns allows for a better choice of pages to replace and makes the idea of specialized buffer replacement policies more attractive in the DBMS environment.

Even more important, being able to predict reference patterns enables the use of a simple and very effective strategy called prefetching of pages. The

Prefetching: IBM DB2 supports both sequential alld list prefeteh (prefetching a list of pages). In general, the prefeteh size is 32 4KB pages, but this can be set by the user. For some sequential type datahaseutilities (e.g., COPY, RUNSTATS), DB2 prefetches up to 64 4KB pages. For a smaller buffer pool (i.e., less than 1000 buffers), the prefetch quantity is adjusted downward to 16 or 8 pages. The prefetch size can be configured by the user; for certain environments, it may be best to prefetch 1000 pages at a time! Sybase ASE supports asynchronous prefetching of up to 256 pages, and uses this capability to reduce latency during indexed access to a table in a range scan. Oracle 8 uses prefetching for sequential scan, retrieving large objects, and certain index scans. Microsoft SQL Server supports prefetching for sequential scan and for scans along the leaf level ofa B+ tree index, and the prefetch size can be adjusted as a scan progresses. SQL Server also uses asynchronous prefetching extensively. Informix supports prefetching with a user-defined prefetch size.

buffer manager can anticipate the next several page requests and fetch the corresponding pages into memory *before* the pages are requested. This strategy has two benefits. First, the pages are available in the buffer pool when they are requested. Second, reading in a contiguous block of pages is much faster than reading the same pages at different times in response to distinct requests. (Review the discussion of disk geometry to appreciate why this is so.) If the pages to be prcfetched are not contiguous, recognizing that several pages need to be fetched can nonetheless lead to faster I/O because an order of retrieval can be chosen for these pages that minimizes seek times and rotational delays.

Incidentally, note that the I/O can typically be done concurrently with CPU computation. Once the prefetch request is issued to the disk, the disk is responsible for reading the requested pages into memory pages and the CPU can continue to do other work.

A DBMS also requires the ability to explicitly *force* a page to disk, that is, to ensure that the copy of the page on disk is updated with the copy in memory. As a related point, a DBMS must be able to ensure that certain pages in the buffer pool are written to disk *before* certain other pages to implement the WAL protocol for crash recovery, as we saw in Section 1.7. Virtual memory implementations in operating systems cannot be relied on to provide such control over when pages are written to disk; the OS command to write a page to disk may be implemented by essentially recording the write request and deferring the actual modification of the disk copy. If the systern crashes in the interim, the effects can be catastrophic for a DBMS. (Crash recovery is discllssed further in Chapter 18.)

Indexes as Files: In Chapter 8, we presented indexes as a way of organizing data records for efficient search. From an implementation standpoint, indexes are just another kind of file, containing records that dil'ect traffic on requests for data records. For example, a tree index is a collection of records organized into one page per node in the tree. It is convenient to actually think of a tree index as *two* files, because it contains two kinds of records: (1) a file of *index entries*, which are records with fields for the index's search key, and fields pointing to a child node, and (2) a file of *data entries*, whose structure depends on the choice of data entry alternative.

9.5 FILES OF RECORDS

We now turn our attention from the way pages are stored on disk and brought into main memory to the way pages are used to store records and organized into logical collections or *files*. Higher levels of the DBMS code treat a page as effectively being a collection of records, ignoring the representation and storage details. In fact, the concept of a collection of records is not limited to the contents of a single page; a file can span several pages. In this section, we consider how a collection of pages can be organized as a file. We discuss how the space on a page can be organized to store a collection of records in Sections 9.6 and 9.7.

9.5.1 Implementing Heap Files

The data in the pages of a heap file is not ordered in any way, and the only guarantee is that one can retrieve all records in the file by repeated requests for the next record. Every record in the file has a unique rid, and every page in a file is of the same size.

Supported operations on a heap file include *CTeatc* and *destroy* files, *insert* a record, *delete* a record with a given rid, *get* a record with a given rid, and *scan* all records in the file. To get or delete a record with a given rid, note that we must be able to find the id of the page containing the record, given the id of the record.

We must keep track of the pages in each heap file to support scans, and we must keep track of pages that contain free space to implement insertion efficiently. \Ve discuss two alternative ways to rnaintain this information. In each of these alternatives, pages must hold two pointers (which are page ids) for file-level bookkeeping in addition to the data.

Linked List of **Pages**

One possibility is to maintain a heap file as a doubly linked list of pages. The DBMS can remember where the first page is located by maintaining a table containing pairs of $\langle heap\_file\_name, page\_Laddr \rangle$ in a known location on disk. We call the first page of the file the *header page.*

An important task is to maintain information about empty slots created by deleting a record from the heap file. This task has two distinct parts: how to keep track of free space within a page and how to keep track of pages that have some free space. We consider the first part in Section 9.6. The second part can be addressed by maintaining a doubly linked list of pages with free space and a doubly linked list of full pages; together, these lists contain *all* pages in the heap file. This organization is illustrated in Figure 9.4; note that each pointer is really a page id.

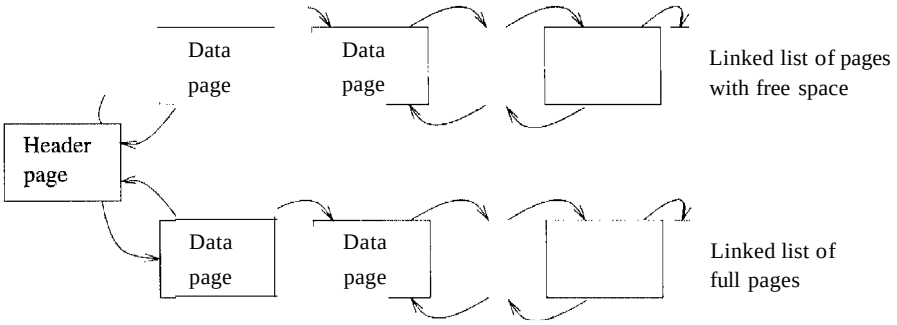


Figure 9.4 Heap File Organization with a Linked List

If a new page is required, it is obtained by making a request to the disk space manager and then added to the list of pages in the file (probably as a page with free space, because it is unlikely that the new record will take up all the space on the page). **If** a page is to be deleted from the heap file, it is removed from the list and the disk space Inanager is told to deallocate it. (Note that the scheme can easily be generalized to allocate or deallocate a sequence of several pages and maintain a doubly linked list of these page sequences.)

One disadvantage of this scheIue is that virtually all pages in a file will be on the free list if records are of variable length, because it is likely that every page has at least a few free bytes. To insert a typical record, we must retrieve and exaInine several pages on the free list before we find one with enough free space. The directory-based heap file organization that we discuss next addresses this problem.

Directory of Pages

An alternative to a linked list of pages is to maintain a **directory of** pages. The DBMS must remember where the first directory page of each heap file is located. The directory is itself a collection of pages and is shown as a linked list in Figure 9.5. (Other organizations are possible for the directory itself, of course.)

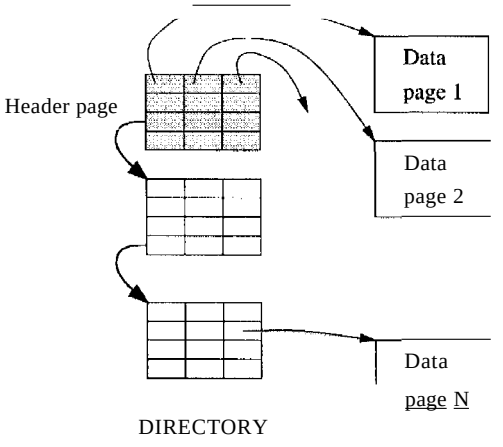


Figure 9.5 Heap File Organization with a Directory

Each directory entry identifies a page (or a sequence of pages) in the heap file. As the heap file grows or shrinks, the number of entries in the directory-and possibly the number of pages in the directory itself--grows or shrinks correspondingly. Note that since each directory entry is quite small in comparison to a typical page, the size of the directory is likely to be very small in comparison to the size of the heap file.

Free space can be managed by maintaining a bit per entry, indicating whether the corresponding page has any free space, or a count per entry, indicating the amount of free space on the page. If the file contains variable-length records, we can examine the free space count for an entry to determine if the record fits on the page pointed to by the entry. Since several entries fit on a directory page, we can efficiently search for a data page with enough space to hold a record to be inserted.

9.6 PAGE FORMATS

The page abstraction is appropriate when dealing with I/O issues, but higher levels of the DBMS see data as a collection of records. In this section, we

Rids in COInmercial Systems: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all implement record ids as a page id and slot number. Sybase ASE uses the following page organization, which is typical: Pages contain a header followed by the rows and a slot array. The header contains the page identity, its allocation state, page free space state, and a timestamp. The slot array is simply a mapping of slot number to page offset.

Oracle 8 and SQL Server use logical record ids rather than page id and slot number in one special case: If a table has a clustered index, then records in the table are identified using the key value for the clustered index. This has the advantage that secondary indexes need not be reorganized if records are moved across pages.

consider how a collection of records can be arranged on a page. We can think of a page as a collection of slots, each of which contains a record. A record is identified by using the pair *(page id, slot number);* this is the record id (rid). (We remark that an alternative way to identify records is to assign each record a unique integer as its rid and maintain a table that lists the page and slot of the corresponding record for each rid. Due to the overhead of maintaining this table, the approach of using *(page id, slot number)* as an rid is more common.)

We now consider some alternative approaches to managing slots on a page. The main considerations are how these approaches support operations such as searching, inserting, or deleting records on a page.

9.6.1 Fixed-Length **Records**

If all records on the page are guaranteed to be of the same length, record slots arc uniform and can be arranged consecutively within a page. At any instant, some slots are occupied by records and others are unoccupied. When a record is inserted into the page, we must locate an empty slot and place the record there. The main issues are how we keep track of empty slots and how we locate all records on a page. The alternatives hinge on how we handle the deletion of a record.

The first alternative is to store records in the first N slots (where N is the number of records on the page); whenever a record is deleted, we move the last record on the page into the vacated slot. This format allows us to locate the ith record on a page by a simple offset calculation, and all empty slots appear together at the end of the page. However, this approach docs not work if there

are external references to the record that is moved (because the rid contains the slot number, which is now changed).

The second alternative is to handle deletions by using an array of bits, one per slot, to keep track of free slot information. Locating records on the page requires scanning the bit array to find slots whose bit is on; when a record is deleted, its bit is turned off. The two alternatives for storing fixed-length records are illustrated in Figure 9.6. Note that in addition to the information about records on the page, a page usually contains additional file-level information (e.g., the id of the next page in the file). The figure does not show this additional information.

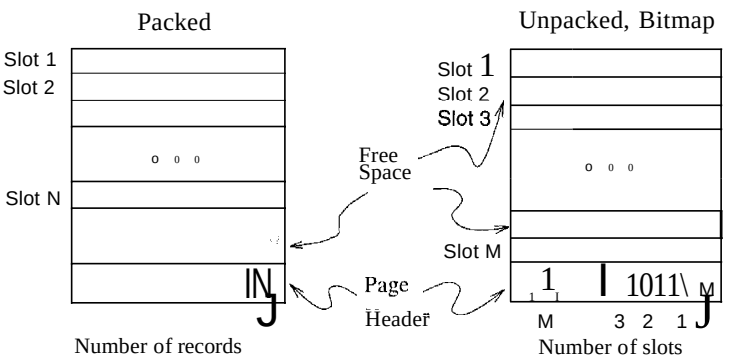


Figure 9.6 Alternative Page Organizations for Fixed-Length Recorcls

The *slotted page* organization described for variable-length records in Section 9.6.2 can also be used for fixed-length records. It becomes attractive if we need to move records around on a page for reasons other than keeping track of space freed by deletions. A typical example is that we want to keep the records on a page sorted (according to the value in some field).

9.6.2 Variable-Length Records

If records are of variable length, then we cannot divide the page into a fixed collection of slots. The problem is that, when a new record is to be inserted, we have to find an empty slot of just the right length----if we use a slot that is too big, we waste space, ancl obviously we cannot use a slot that is smaller than the record length. Therefore, when a record is inserted, we must allocate just the right amount of space for it, and when a record is deleted, we must move records to fill the hole created by the deletion, to ensure that all the free space on the page is contiguous. Therefore, the ability to move records on a page becomes very important.

The most flexible organization for variable-length records is to maintain a directory of slots for each page, with a *(record offset, record length)* pair per slot. The first component *(record offset)* is a 'pointer' to the record, as shown in Figure 9.7; it is the offset in bytes from the start of the data area on the page to the start of the record, Deletion is readily accomplished by setting the record offset to -1. Records can be moved around on the page because the rid, which is the page number and slot number (that is, position in the directory), does not change when the record is moved; only the record offset stored in the slot changes.

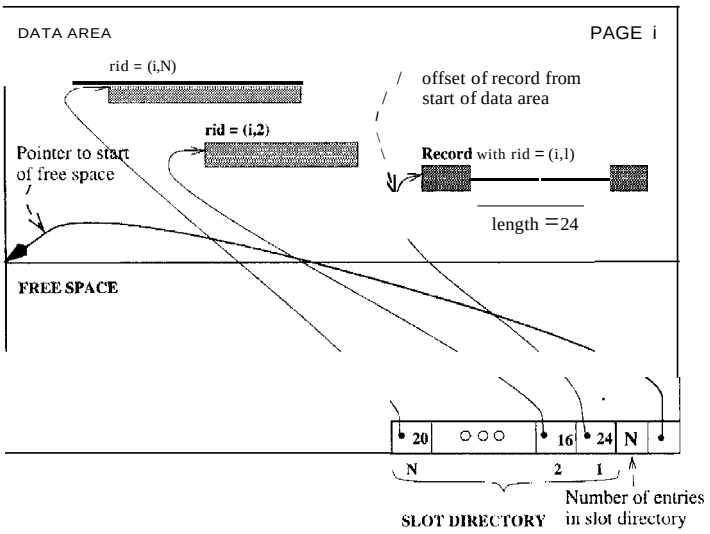


Figure 9.7 Page Organization for Variable-Length R.ecords

The space available for new records must be managed carefully because the page is not preformatted into slots. One way to manage free space is to maintain a pointer (that is, offset from the start of the data area on the page) that indicates the start of the free space area. When a new record is too large to fit into the remaining free space, we have to move records on the page to reclairn the space freed by records deleted earlier. The idea is to ensure that, after reorganization, all records appear in contiguous order, followed by the available free space.

A subtle point to be noted is that the slot for a deleted record cannot always be removed from the slot directory, because slot numbers are used to identify records---by deleting a slot, we change (decrement) the slot number of subsequent slots in the slot directory, and thereby change the rid of records pointed to by subsequent slots. The only way to remove slots from the slot directory is to remove the last slot if the record that it points to is deleted. However, when

a record is inserted, the slot directory should be scanned for an element that currently does not point to any record, and this slot should be used for the new record. A new slot is added to the slot directory only if all existing slots point to records. If inserts are much more common than deletes (as is typically the case), the number of entries in the slot directory is likely to be very close to the actual number of records on the page.

This organization is also useful for fixed-length records if we need to move them around frequently; for example, when we want to maintain them in some sorted order. Indeed, when all records are the same length, instead of storing this common length information in the slot for each record, we can store it once in the system catalog.

In some special situations (e.g., the internal pages of a B+ tree, which we discuss in Chapter 10), we lIlay not care about changing the rid of a record. In this case, the slot directory can be compacted after every record deletion; this strategy guarantees that the number of entries in the slot directory is the same as the number of records on the page. If we do not care about modifying rids, we can also sort records on a page in an efficient manner by simply moving slot entries rather than actual records, which are likely to be much larger than slot entries.

A simple variation on the slotted organization is to maintain only record offsets in the slots. For variable-length records, the length is then stored with the record (say, in the first bytes). This variation makes the slot directory structure for pages with fixed-length records the sallle as for pages with variable-length records.

9.7 RECORD FORMATS

In this section, we discuss how to organize fields within a record. While choosing a way to organize the fields of a record, we must take into account whether the fields of the record are of fixed or variable length and consider the cost of various operations on the record, including retrieval and modification of fields.

Before discussing record fonnats, we note that in addition to storing individual records, information conllnon to all records of a given record type (such as the number of fields and field types) is stored in the system catalog, which can be thought of as a description of the contents of a database, maintained by the DBMS (Section 12.1). This avoids repeated storage of the same information with each record of a given type.

Record Formats **in** Commercial Aystems: In IBM DB2, fixed-length fields are at fixed offsets from the beginning of the record. Variable-length fields have offset and length in the fixed offset part of the record, and the fields themselves follow the fixed-length part of the record. Informix, Microsoft SQL Server, and Sybase ASE use the same organization with minor variations. In Oracle 8, records are structured as if all fields are potentially of variable length; a record is a sequence of length-data pairs, with a special length value used to denote a *null* value.

9.7.1 Fixed-Length Records

In a fixed-length record, each field has a fixed length (that is, the value in this field is of the same length in all records), and the number of fields is also fixed. The fields of such a record can be stored consecutively, and, given the address of the record, the address of a particular field can be calculated using information about the lengths of preceding fields, which is available in the system catalog. This record organization is illustrated in Figure 9.8.

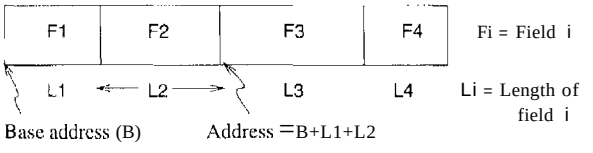


Figure 9.8 Organization of Records with Fixed-Length Fields

9.7.2 Variable-Length Records

In the relational model, every record in a relation contains the same number of fields. **If** the number of fields is fixed, a record is of variable length only because some of its fields are of variable length.

One possible orga,nizatioll is to store fields consecutively, separated by delimiters (which are special characters that do not appear in the data itself). This organization requires a scan of the record to locate a desired field.

An alternative is to reserve some space at the beginning of a record for use as an array of integer offsets—the ith integer in this array is the starting address of the ith field value relative to the start of the record. Note that we also store an offset to the end of the record; this offset is needed to recognize where the last field ends. Both alternatives are illustrated in Figure 9.9.

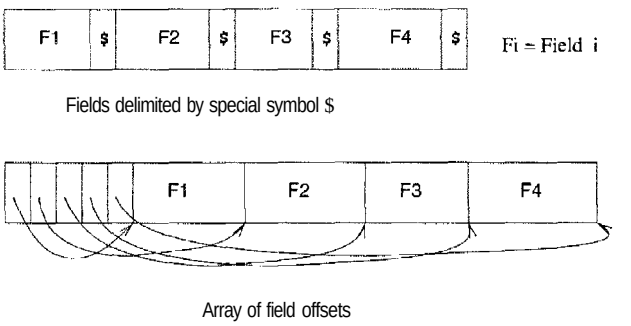


Figure 9.9 Alternative Record Organizations for Variable-Length Fields

The second approach is typically superior. For the overhead of the offset array, we get direct access to any field. We also get a clean way to deal with **null** values. A *null* value is a special value used to denote that the value for a field is unavailable or inapplicable. If a field contains a *null* value, the pointer to the end of the field is set to be the same as the pointer to the beginning of the field. That is, no space is used for representing the *null* value, and a comparison of the pointers to the beginning and the end of the field is used to determine that the value in the field is *null*.

Variable-length record formats can obviously be used to store fixed-length records as well; sometimes, the extra overhead is justified by the added flexibility, because issues such as supporting *n'ull* values and adding fields to a recorcl type arise with fixed-length records as well.

I-laving variable-length fields in a record can raise some subtle issues, especially when a record is modified.

- Modifying a field may cause it to grow, which requires us to shift all subsequent fields to make space for the modification in all three record formats just presentcel.
- A modified record may no longer fit into the space remaining on its page. If so, it may have to be moved to another page. If riels, which are used to 'point' to a record, include the page number (see Section 9.6), moving a record to 'another page causes a problem. We may have to leave a 'forwarding address' on this page identifying the ne"v location of the record. And to ensure that space is ahvays available for this forwarding address, we would have to allocate some minimum space for each record, regardless of its length.

Large Records in Real Systems: In Sybase ASE, a record can be at most 1962 bytes. This limit is set by the 2KB log page size, since records are not allowed to be larger than a page. The exceptions to this rule are BLOBs and CLOBs, which consist of a set of bidirectionally linked pages. IBM DB2 and Microsoft SQL Server also do not allow records to span pages, although large objects are allowed to span pages and are handled separately from other data types. In DB2, record size is limited only by the page size; in SQL Server, a record can be at most 8KB, excluding LOBs. Informix and Oracle 8 allow records to span pages. Informix allows records to be at most 32KB, while Oracle has no maximum record size; large records are organized as a singly directed list.

- A record may grow so large that it no longer fits on *anyone* page. We have to deal with this condition by breaking a record into smaller records. The smaller records could be chained together-part of each smaller record is a pointer to the next record in the chain---to enable retrieval of the entire original record.

9.8 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Explain the term *memory hierarchy.* What are the differences between primary, secondary, and tertiary storage? Give examples of each. Which of these is *volatile*, and which are *persistent?* Why is persistent storage more important for a DBMS than, say, a program that generates prime numbers? (Section 9.1)
- Why are disks used so widely in a DBMS? What are their advantages over main memory and tapes? What are their relative disadvantages? (Section 9.1.1)
- What is a *disk block* or *page?* How are blocks arranged in a disk? How does this affect the time to access a block? Discuss *seek time*, *rotational delay*, and *transfer time.* (Section 9.1.1)
- Explain how careful placement of pages on the disk to exploit the geometry of a disk can minimize the seek time and rotational delay when pages are read sequentially. (Section 9.1.2)
- Explain what a RAID systenl is and how it improves performance and reliability. Discuss *striping* and its impact on performance and *redundancy* and its irnpact on reliability. What are the trade-offs between reliability

and performance in the different RAID organizations called *RAID levels'?* (Section 9.2)

- What is the role of the DBMS *disk space manager?* Why do database systems not rely on the operating system instead? (Section 9.3)
- Why does every page request in a DBMS go through the buffer manager? What is the *buffer poor?* What is the difference between a *frame* in a buffer pool, a *page* in a file, and a *block* on a disk? (Section 9.4)
- What information does the buffer manager maintain for each page in the buffer pool? ·What information is maintained for each frame? What is the significance of *pin_count* and the *dirty* flag for a page? Under what conditions can a page in the pool be *replaced?* Under what conditions must a replaced page be written back to disk? (Section 9.4)
- Why does the buffer manager have to replace pages in the buffer pool? How is a page chosen for replacement? What is *sequential flooding*, and what replacement policy causes it? (Section 9.4.1)
- A DBMS buffer manager can often predict the access pattern for disk pages. How does it utilize this ability to minimize I/O costs? Discuss *prefetching.* What is *forcing*, and why is it required to support the write-ahead log protocol in a DBMS? In light of these points, explain why database systems reimplement many services provided by operating systems. (Section 9.4.2)
- Why is the abstraction of a *file of records* important? How is the software in a DBMS layered to take advantage of this? (Section 9.5)
- What is a *heap file?* How are pages organized in a heap file? Discuss list versus directory organizations. (Section 9.5.1)
- Describe how records are arranged on a page. What is a *slot*, and how are slots used to identify records? How do slots ena.ble us to move records on a page withont altering the record's identifier? ·What arc the differences in page organizations for fixed-length and variable-length records? (Section 9.6)
- ·What are the differences in how fields are arranged within fixed-length and variable-length records? For variable-length records, explain how the array of offsets organization provides direct access to a specific field and supports *null* values. (Section 9.7)

EXERCISES

Exercise 9.1 What is the most important difference between a disk and a tape?

Exercise 9.2 Explain the terms *seek time, rotational delay,* and *transfer time.*

Exercise 9.3 Both disks and main memory support direct access to any desired location (page). On average, main memory accesses are faster, of course. What is the other important difference (from the perspective of the time required to access a desired page)?

Exercise 9.4 If you have a large file that is frequently scanned sequentially, explain how you would store the pages in the file on a disk.

Exercise 9.5 Consider a disk with a sector size of 512 bytes, 2000 tracks per surface, 50 sectors per track, five double-sided platters, and average seek time of 10 msec.

1. What is the capacity of a track in bytes? What is the capacity of each surface? What is the capacity of the disk?
2. How many cylinders does the disk have?
3. Give examples of valid block sizes. Is 256 bytes a valid block size? 2048? 51,200?
4. If the disk platters rotate at 5400 rpm (revolutions per minute), what is the maximum rotational delay?
5. If one track of data can be transferred per revolution, what is the transfer rate?

Exercise 9.6 Consider again the disk specifications from Exercise 9.5 and suppose that a block size of 1024 bytes is chosen. Suppose that a file containing 100,000 records of 100 bytes each is to be stored on such a disk and that no record is allowed to span two blocks.

1. How many records fit onto a block?
2. How many blocks are required to store the entire file? If the file is arranged sequentially on disk, how lllallY surfaces are needed?
3. How many records of 100 bytes each can be stored using this disk?
4. If pages are stored sequentially on disk, with page 1 on block 1 of track 1, what page is stored on block 1 of track 1 on the next disk surface? How would your answer change if the disk were capable of reading and writing from all heads in parallel?
5. What titne is required to read a file containing 100,000 records of 100 bytes each sequentially? Again, how \vould your answer change if the disk were capable of reading/writing from all heads in parallel (and the data was arranged optimally)?
6. What is the time required to read a file containing 100,000 records of 100 bytes each in a random order? To read a record, the block containing the recOl'd has to be fetched from disk. Assume that each block request incurs the average seek time and rotational delay.

Exercise 9.7 Explain what the buffer manager JIms! do to process a read request for a page. \Vhat happens if the requested page is in the pool but not pinned?

Exercise 9.8 When does a buffer manager write a page to disk?

Exercise 9.9 What does it mean to say that a page is *pinned* in the buffer pool? Who is responsible for pinning pages? \Vho is responsible for unpinning pages?

Exercise 9.10 'When a page in the bulTer pool is modified, how does the DBMS ensure that this change is propagated to disk? (Explain the role of the buffer manager as well as the modifier of the page.)

Exercise 9.11 \Vhat happens if a page is requested when all pages in the buffer pool are dirty?

Exercise 9.12 \Vhat is *sequential flooding* of the buffer pool?

Exercise 9.13 Name an important capability of a DBMS buffer manager that is not supported by a typical operating system's buffer manager.

Exercise 9.14 Explain the term *prefetching.* Why is it important?

Exercise 9.15 Modern disks often have their own main memory caches, typically about 1 MB, and use this to prefetch pages. The rationale for this technique is the empirical observation that, if a disk page is requested by some (not necessarily database!) application, 80% of the time the next page is requested as well. So the disk gambles by reading ahead.

1. Give a nontechnical reason that a DBMS may not want to rely on prefetching controlled by the disk.
2. Explain the impact on the disk's cache of several queries running concurrently, each scanning a different file.
3. Is this problem addressed by the DBMS buffer manager prefetching pages? Explain.
4. Modern disks support *segmented caches*, with about four to six segments, each of which is used to cache pages from a different file. Does this technique help, with respect to the preceding problem? Given this technique, does it matter whether the DBMS buffer manager also does prefetching?

Exercise 9.16 Describe two possible record formats. What are the trade-offs between them?

Exercise 9.17 Describe two possible page formats. What are the trade-offs between them?

Exercise 9.18 Consider the page format for variable-length records that uses a slot directory.

1. One approach to managing the slot directory is to use a maximum size (i.e., a maximum number of slots) and allocate the directory array when the page is created. Discuss the pros and cons of this approach with respect to the approach discussed in the text.
2. Suggest a modification to this page format that would allow us to sort records (according to the value in some field) without moving records and without changing the record ids.

Exercise 9.19 Consider the two internal organizations for heap files (using lists of pages and a directory of pages) discussed in the text.

1. Describe them briefly and explain the trade-offs. Which organization would you choose if records are variable in length?
2. Can you suggest a single page format to implement both internal file organizations'?

Exercise 9.20 Consider a list-based organizat.ion of the pages in a heap file in which two lists are maintained: a list of *all* pages in the file and a list of all pages with free space. In contrast, the list-based organizatioll discussed in the text maintains a list of full pages and a list of pages with free space.

1. What are the trade-offs, if any? Is one of them clearly superior?
2. For each of these organizations, describe a suitable page format.

Exercise **9.21** Modern disk drives store more sectors on the outer tracks than the inner tracks. Since the rotation speed is constant, the sequential data transfer rate is also higher on the outer tracks. The seek time and rotational delay are unchanged. Given this information, explain good strategies for placing files with the following kinds of access patterns:

1. Frequent, random accesses to a small file (e.g., catalog relations).
2. Sequential scans of a large file (e.g., selection from a relation with no index).
3. Random accesses to a large file via an index (e.g., selection from a relation via the index).
4. Sequential scans of a small file.

Exercise **9.22** Why do frames in the buffer pool have a pin count instead of a pin flag?

PROJECT-BASED EXERCISES

Exercise **9.23** Study the public interfaces for the disk space manager, the buffer manager, and the heap file layer in Minibase.

1. Are heap files with variable-length records supported?
2. What page format is used in Minibase heap files?
3. What happens if you insert a record whose length is greater than the page size?
4. How is free space handled in Minibase?

BIBLIOGRAPHIC NOTES

Salzberg [648] and Wiederhold [776] discuss secondary storage devices and file organizations in detail.

RAID was originally proposed by Patterson, Gibson, and Katz [587]. The article by Chen et al. provides an excellent survey of RAID [171] . Books about RAID include Gibson's dissertation [317] and the publications from the RAID Advisory Board [605].

The design and implementation of storage managers is discussed in [65, 133, 219, 477, 718]. With the exception of [219], these systems emphasize *extensibility*, anel the papers contain much of interest from that stanelpoint as well. Other papers that cover storage management issues in the context of significant implemented prototype systems are [480] and [588]. The Dali storage Inanager, which is optimized for main memory databases, is described in [406]. Three techniques for ilnplementing long fields are compared in [96]. The impact of processor cache misses 011 DBMS performallce has received attention lately, as complex queries have become increasingly CPU-intensive. [33] studies this issue, and shows that performance can be significantly improved by using a new arrangement of records within a page, in which records on a page are stored in a column-oriented format (all field values for the first attribute followed by values for the second attribute, etc.).

Stonebraker discusses operating systems issues in the context of databases in [715]. Several buffer management policies for database systems are compared in [181]. Buffer management is also studied in [119, 169, 2G1, 235].

10

TREE-STRUCTURED INDEXING

- What is the intuition behind tree-structured indexes? Why are they good for range selections?
- How does an ISAM index handle search, insert, and delete?
- How does a B+ tree index handle search, insert, and delete?
- What is the impact of duplicate key values on index implementation'?
- What is key compression, and why is it important?
- What is bulk-loading, and why is it important?
- What happens to record identifiers when dynamic indexes are updated? How does this affect clustered indexes?

Key concepts: ISAM, static indexes, overflow pages, locking issues; B+ trees, dynamic indexes, balance, sequence sets, node format; B+ tree insert operation, node splits, delete operation, merge versus redistribution, minimum occupancy; duplicates, overflow pages, including rids in search keys; key compression; bulk-loading; effects of splits on rids in clustered indexes.

One that would have the fruit must climb the tree.

Thomas Fuller

We now consider two index data structures, called ISAM and B+ trees, based on tree organizations. These structures provide efficient support for range searches, including sorted file scans as a special case. Unlike sorted files, these

index structures support efficient insertion and deletion. They also provide support for equality selections, although they are not as efficient in this case as hash-based indexes, which are discussed in Chapter 11.

An ISAJVI[1] tree is a static index structure that is effective when the file is not frequently updated, but it is unsuitable for files that grow and shrink a lot. \Ve discuss ISAM in Section 10.2. The B+ tree is a dynamic structure that adjusts to changes in the file gracefully. It is the most widely used index structure because it adjusts well to changes and supports both equality and range queries. We introduce B+ trees in Section 10.3. We cover B+ trees in detail in the remaining sections. Section 10.3.1 describes the format of a tree node. Section *IOA* considers how to search for records by using a B+ tree index. Section 10.5 presents the algorithm for inserting records into a B+ tree, and Section 10.6 presents the deletion algorithm. Section 10.7 discusses how duplicates are handled. We conclude with a discussion of some practical issues concerning B+ trees in Section 10.8.

Notation: In the ISAM and B+ tree structures, leaf pages contain *data entries*, according to the terminology introduced in Chapter 8. For convenience, we denote a data entry with search key value k as $k*$. Non-leaf pages conta.in *index entries* of the form *(search key value, page id)* and are used to direct the sea.rch for a desired data entry (which is stored in some leaf). We often simply use *entr'Y* where the context makes the nature of the entry (index or data) clear.

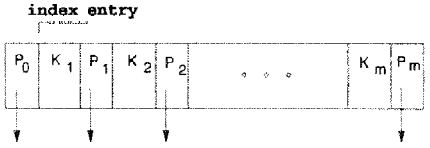
10.1 INTUITION FOR TREE INDEXES

Consider a file of Students recorcls sorted by *gpa*. To answer a range selection such as "Find all students with a gpa higher than 3.0," we must identify the first such student by doing a binary search of the file and then scan the file from that point on. If the file is large, the initial binary search can be quite expensive, since cost is proportional to the number of pages fetched; can we improve upon this method?

OIle idea is to create a second file with OIle record per page in the original (data) file, of the form *(first key on page, pointer to page)*, again sortecl by the key attribute (which is *gpa* in our example). The format of a page in the second *index* file is illustrated in Figure 10.1.

We refer to pairs of the form *(key, pointer)* as *index entries* or just *entries* when the context is clear. Note that each index page contains OIle pointer more than

[1] ISAM stands for Indexed Sequential Access Method.



FigUl'e 10.1 Format of an Index Page

the number of keys—each key serves as a *separator* for the contents of the pages pointed to by the pointers to its left and right.

The simple index file data structure is illustrated in Figure 10.2.

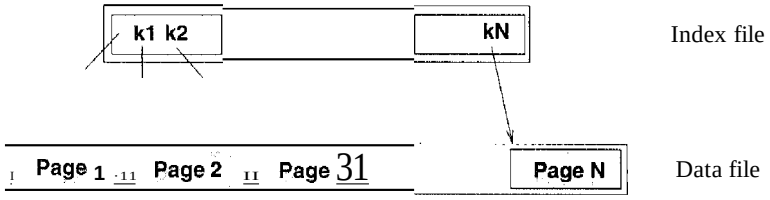


Figure 10.2 One-Level Index Structure

We can do a binary search of the index file to identify the page containing the first key *(gpo.)* value that satisfies the range selection (in our example, the first student with *gpa* over 3.0) and follow the pointer to the page containing the first data. record with that key value. We can then scan the data file sequentially from that point on to retrieve other qualifying records. This example uses the index to find the first data page containing a Students record with *gpa* greater than 3.0, and the data file is scanned from that point on to retrieve other such Students records.

Because the size of an entry in the index file (key value and page icl) is likely to be much smaller than the size of a page, and only one such entry exists per page of the data file, the index file is likely to be much smaller than the data file; therefore, a binary search of the index file is much faster than a binary search of the data file. However, a binary search of the index file could still be fairly expensive, and the index file is typically still large enough to make inserts and deletes expensive.

The potential large size of the index file motivates the tree indexing idea: Why not apply the previous step of building an auxiliary structure all the collection of *index* records and so on recursively until the smallest auxiliary structure fits on one page? This repeated construction of a one-level index leads to a tree structure with several levels of non-leaf pages.

As we observed in Section 8.3.2, the power of the approach comes from the fact that locating a record (given a search key value) involves a traversal from the root to a leaf, with one I/O (at most; some pages, e.g.) the root, are likely to be in the buffer pool) per level. Given the typical fan-out value (over 100), trees rarely have more than 3–4 levels.

The next issue to consider is how the tree structure can handle inserts and deletes of data entries. Two distinct approaches have been used, leading to the ISAM and B+ tree data structures, which we discuss in subsequent sections.

10.2 INDEXED SEQUENTIAL ACCESS METHOD (ISAM)

The ISAM data structure is illustrated in Figure 10.3. The data entries of the ISAM index are in the leaf pages of the tree and additional *overflow* pages chained to some leaf page. Database systems carefully organize the layout of pages so that page boundaries correspond closely to the physical characteristics of the underlying storage device. The ISAM structure is completely static (except for the overflow pages, of which it is hoped, there will be few) and facilitates such low-level optimizations.

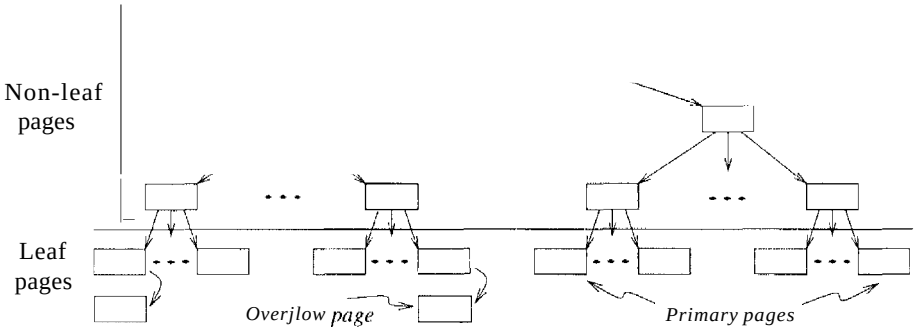


Figure 10.3 ISAM Index Structure

Each tree node is a disk page, and all the data resides in the leaf pages. This corresponds to an index that uses Alternative (1) for data entries, in terms of the alternatives described in Chapter 8; we can create an index with Alternative (2) by storing the data records in a separate file and storing $\langle key, rid \rangle$ pairs in the leaf pages of the ISAM index. When the file is created, all leaf pages are allocated sequentially and sorted on the search key value. (If Alternative (2) or (3) is used, the data records are created and sorted before allocating the leaf pages of the ISAM index.) The non-leaf level pages are then allocated. If there are several inserts to the file subsequently, so that more entries are inserted into a leaf than will fit onto a single page, additional pages are needed because the

index structure is static. These additional pages are allocated from an overflow area. The allocation of pages is illustrated in Figure 10.4.

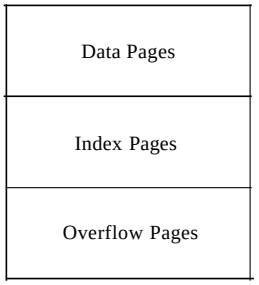


Figure 10.4 Page Allocation in ISAM

The basic operations of insertion, deletion, and search are all quite straightforward. For an equality selection search, we start at the root node and determine which subtree to search by comparing the value in the search field of the given record with the key values in the node. (The search algorithm is identical to that for a B+ tree; we present this algorithm in more detail later.) For a range query, the starting point in the data (or leaf) level is determined similarly, and data pages are then retrieved sequentially. For inserts and deletes, the appropriate page is determined as for a search, and the record is inserted or deleted with overflow pages added if necessary.

The following example illustrates the ISAM index structure. Consider the tree shown in Figure 10.5. All searches begin at the root. For example, to locate a record with the key value 27, we start at the root and follow the left pointer, since 27 < 40. We then follow the middle pointer, since 20 <= 27 < 33. For a range sea,rch, we find the first qualifying data entry as for an equality selection and then retrieve primary leaf pages sequentially (also retrieving overflow pages as needed by following pointers from the primary pages). The primary leaf pages are assumed to be allocated sequentially this assumption is reasonable because the number of such pages is known when the tree is created and does not change subsequently under inserts and deletes-and so no 'next leaf page' pointers are needed.

We assume that each leaf page can contain two entries. If we now insert a record with key value 23, the entry 23* belongs in the second data page, which already contains 20* and 27* and has no more space. We deal with this situation by adding an *overflow* page and putting 23* in.the overflow page. Chains of overflow pages can easily develop. For instance, inserting 48*, 41*, and 42* leads to an overflow chain of two pages. The tree of Figure 10.5 with all these insertions is shown ill Figure 10.6.

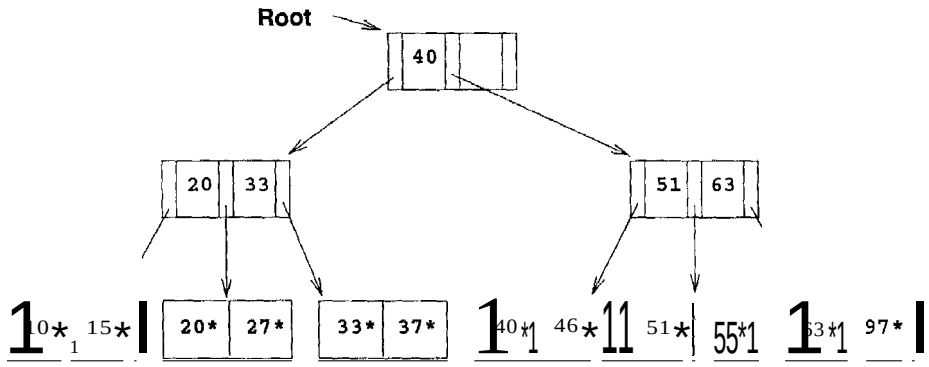


Figure 10.5 Sample ISAM Tree

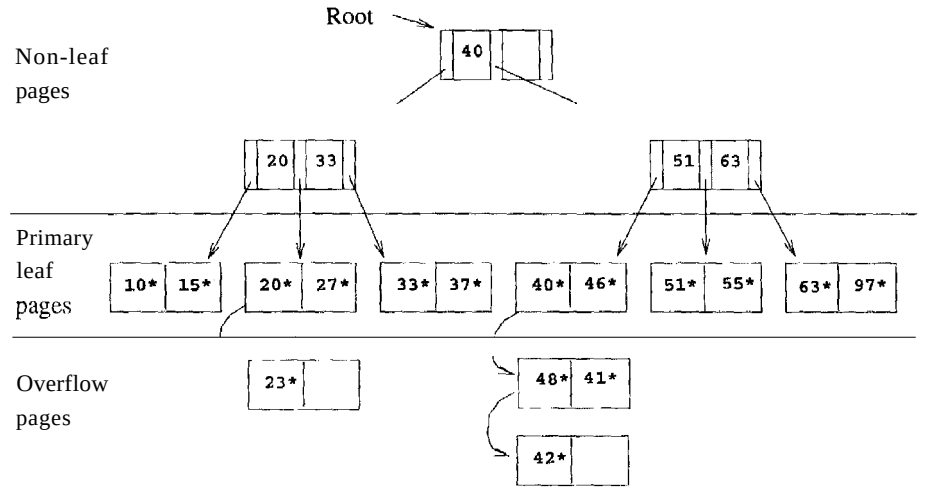


Figure 10.6 ISAM Tree after Inserts

The deletion of an entry $k*$ is handled by simply removing the entry. If this entry is on an overflow page and the overflow page becomes empty, the page can be removed. If the entry is on a primary page and deletion makes the primary page empty, the simplest approach is to simply leave the empty primary page as it is; it serves as a placeholder for future insertions (and possibly lloll-empty overflow pages, because we do not move records from the overflow pages to the primary page when deletions on the primary page create space). Thus, the number of primary leaf pages is fixed at file creation time.

10.2.1 Overflow Pages, Locking Considerations

Note that, once the ISAM file is created, inserts and deletes affect only the contents of leaf pages. A consequence of this design is that long overflow chains could develop if a number of inserts are made to the same leaf. These chains can significantly affect the time to retrieve a record because the overflow chain has to be searched as well when the search gets to this leaf. (Although data in the overflow chain can be kept sorted, it usually is not, to make inserts fast.) To alleviate this problem, the tree is initially created so that about 20 percent of each page is free. However, once the free space is filled in with inserted records, unless space is freed again through deletes, overflow chains can be eliminated only by a complete reorganization of the file.

The fact that only leaf pages are modified also has an important advantage with respect to concurrent access. When a page is accessed, it is typically 'locked' by the requestor to ensure that it is not concurrently modified by other users of the page. To modify a page, it must be locked in 'exclusive' mode, which is permitted only when no one else holds a lock on the page. Locking can lead to queues of users (*transactions*, to be more precise) waiting to get access to a page. Queues can be a significant performance bottleneck, especially for heavily accessed pages near the root of an index structure. In the ISAM structure, since we know that index-level pages are never modified, we can safely omit the locking step. Not locking index-level pages is an important advantage of ISAM over a dynamic structure like a B+ tree. If the data distribution and size are relatively static, which means overflow chains are rare, ISAM might be preferable to B+ trees due to this advantage.

10.3 B+ TREES: A DYNAMIC INDEX STRUCTURE

A static structure such as the ISAM index suffers from the problem that long overflow chains can develop as the file grows, leading to poor performance. This problem motivated the development of more flexible, dynamic structures that adjust gracefully to inserts and deletes. The B+ tree search structure, which is widely llsed, is a balanced tree in which the internal nodes direct the search

and the leaf nodes contain the data entries. Since the tree structure grows and shrinks dynamically, it is not feasible to allocate the leaf pages sequentially as in ISAM, where the set of primary leaf pages was static. To retrieve all leaf pages efficiently, we have to link them using page pointers. By organizing them into a doubly linked list, we can easily traverse the sequence of leaf pages (sometimes called the sequence set) in either direction. This structure is illustrated in Figure 10.7.[2]

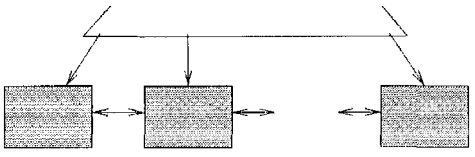


Figure 10.7 Structure of a B+ Tree

The following are some of the main characteristics of a B+ tree:

- Operations (insert, delete) on the tree keep it balanced.
- A minimum occupancy of 50 percent is guaranteed for each node except the root if the deletion algorithm discussed in Section 10.6 is implemented. However, deletion is often implemented by simply locating the data entry and removing it, without adjusting the tree as needed to guarantee the 50 percent occupancy, because files typically grow rather than shrink.
- Searching for a record requires just a traversal from the root to the appropriate leaf. We refer to the length of a path from the root to a leaf—any leaf, because the tree is balanced—as the height of the tree. For example, a tree with only a leaf level and a single index level, such as the tree shown in Figure 10.9, has height 1, and a tree that has only the root node has height 0. Because of high fan-out, the height of a B+ tree is rarely more than 3 or 4.

We will study B+ trees in which every node contains m entries, where $d \leq m \leq 2d$. The value d is a parameter of the B+ tree, called the order of the

[2] If the tree is created by *bulk_looding* (see Section 10.8.2) an existing data set, the sequence set can be made physically sequential, but this physical ordering is gradually destroyed as new data is added and deleted over time.

tree, and is a measure of the capacity of a tree node. The root node is the only exception to this requirement on the number of entries; for the root, it is simply required that $1 \leq m \leq 2d$.

If a file of records is updated frequently and sorted access is important, maintaining a B+ tree index with data records stored as data entries is almost always superior to maintaining a sorted file. For the space overhead of storing the index entries, we obtain all the advantages of a sorted file plus efficient insertion and deletion algorithms. B+ trees typically maintain 67 percent space occupancy. B+ trees are usually also preferable to ISAM indexing because inserts are handled gracefully without overflow chains. However, if the dataset size and distribution remain fairly static, overflow chains may not be a major problem. In this case, two factors favor ISAM: the leaf pages are allocated in sequence (making scans over a large range more efficient than in a B+ tree, in which pages are likely to get out of sequence on disk over time, even if they were in sequence after bulk-loading), and the locking overhead ofISAM is lower than that for B+ trees. As a general rule, however, B+ trees are likely to perform better than ISAM.

10.3.1 Format of a Node

The format of a node is the same as for ISAM and is shown in Figure 10.1. Non-leaf nodes with m *index entries* contain $m+1$ pointers to children. Pointer *Pi* points to a subtree in which all key va.lues K are such that $K_i \leq K < \mathrm{K_i{+}_1}$. As special cases, *Po* points to a tree in which all key values are less than Kl' and P_m points to a tree in which all key values are greater than or equal to K_m. For leaf nodes, entries arc denoted as *k**, as usual. Just as in ISAM, leaf nodes (and *only* leaf nodes!) contain *data entries.* In the common case that Alternative (2) or (3) is used, leaf entries are *(K,I(K)* $\rangle$ pairs, just like non-leaf entries. Regardless of the alternative chosen for leaf entries, the leaf pages are chained together in a doubly linked list. Thus, the leaves form a sequence, which can be used to answer range queries efficiently.

The reader should carefully consider how such a node organization can be achieved using the record formats presented in Section 9.7; after all, each key pointer pair can be thought of as a record. If the field being indexed is of fixed length, these index entries will be of fixed length; otherwise, we have variable-length records. In either case the B+ tree can itself be viewed as a file of records. If the leaf pages do not contain the actual data records, then the 13+ tree is indeed a file of records that is distinct from the file that contains the data. If the leaf pages contain data records, then a file contains the 13+ tree as well as the data.

10.4 SEARCH

The algorithm for search finds the leaf node in which a given data entry belongs. A pseudocode sketch of the algorithm is given in Figure 10.8. We use the notation **ptr* to denote the value pointed to by a pointer variable *ptr* and & *(value)* to denote the address of *value*. Note that finding *i* in *tTcc_seaTch* requires us to search within the node, which can be done with either a linear search or a binary search (e.g., depending on the number of entries in the node).

In discussing the search, insertion, and deletion algorithms for B+ trees, we assume that there are no *duplicates*. That is, no two data entries are allowed to have the same key value. Of course, duplicates arise whenever the search key does not contain a candidate key and must be dealt with in practice. We consider how duplicates can be handled in Section 10.7.

```
fune find (search key value K) returns nodepointer
// Given a search key value, finds its leaf node
return tree_search(root, K);                    // searches from root
endfune

fune tree_search (nodepointer, search key value K) returns nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
     if K < K_1 then return tree_search(P_0, K);
     else,
          if K ≥ K_m then return tree_search(P_m, K);    // 171 = # entries
          else,
               find i such that K_i ≤ K < K_{i+1};
               return tree_search(P_i, K)
endfune
```

Figure 10.8 Algorithm for Search

Consider the sample B+ tree shown in Figure 10.9. This B+ tree is of order d=2. That is, each node contains between 2 and 4 entries. Each non--leaf entry is a $\langle key\ value, nodepointer \rangle$ pair; at the leaf level, the entries are data records that we denote by $k*$. To search for entry 5*, we follow the left-most child pointer, since 5 < 13. To search for the entries 14* or 15*, we follow the second pointer, since $13 \leq 14 < 17$, and $13 \leq 15 < 17$. (We do not find 15* on the appropriate leaf and can conclude that it is not present in the tree.) To find 24*, we follow the fourth child pointer, since $24 \leq 24 < 30$.

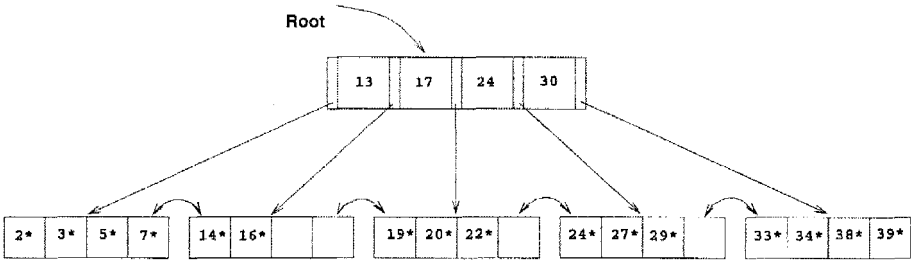


Figure 10.9 Example of a B+ Tree, Order d=2

10.5 INSERT

The algorithm for insertion takes an entry, finds the leaf node where it belongs, and inserts it there. Pseudocode for the B+ tree insertion algorithm is given in Figure HUG. The basic idea behind the algorithm is that we recursively insert the entry by calling the insert algorithm on the appropriate child node. Usually, this procedure results in going down to the leaf node where the entry belongs, placing the entry there, and returning all the way back to the root node. Occasionally a node is full and it must be split. When the node is split, an entry pointing to the node created by the split must be inserted into its parent; this entry is pointed to by the pointer variable *newchildentry*. If the (old) root is split, a new root node is created and the height of the tree increases by 1.

To illustrate insertion, let us continue with the sample tree shown in Figure 10.9. If we insert entry 8*, it belongs in the left-most leaf, which is already full. This insertion causes a split of the leaf page; the split pages are shown in Figure 10.11. The tree must now be adjusted to take the new leaf page into account, so we insert an entry consisting of the pair (5, *pointer to new page)* into the parent node. Note how the key 5, which discriminates between the split leaf page and its newly created sibling, is 'copied up.' We cannot just 'push up' 5, because every data entry must appear in a leaf page.

Since the parent node is also full, another split occurs. In general we have to split a non-leaf node when it is full, containing $2d$ keys and $2d+1$ pointers, and we have to add another index entry to account for a child split. We now have $2d+1$ keys and $2d+2$ pointers, yielding two minimally full non-leaf nodes, each containing d keys and $d+1$ pointers, and an extra key, which we choose to be the 'middle' key. This key and a pointer to the second non-leaf node constitute an index entry that must be inserted into the parent of the split non-leaf node. The middle key is thus 'pushed up' the tree, in contrast to the case for a split of a leaf page.

```
proc inseTt (nodepointel', entry, newchildentry)
// InseTts entry into subtree with TOot '*nodepointer'; degree is d;
//'newchildentry' null initially, and null on return unless child is split

if *nodepointer is a non-leaf node, say N,
    find'i such that K_i ≤ entry's key value < J(i+1;    // choose subtree
    insert(P_i, entry, newchildentry);          // recurs'ively, insert entry
    if newchildentry is null, return;          // usual case; didn't split child
    else,                  // we split child, must insert *newchildentry in N
        if N has space,                                        // usual case
            put *newchildentry on it, set newchildentry to null, return;
        else,                  // note difference wrt splitting of leaf page!
            split N:          // 2d + 1 key values and 2d + 2 nodepointers
            first d key values and d + 1 nodepointers stay,
            last d keys and d + 1 pointers move to new node, N2;
            // *newchildentry set to guide searches between N and N2
            newchildentry = & ((smallest key value on N2,
                                pointer to N2));
            if N is the root,                  // root node was just split
                create new node with (pointer to N, *newchildentry);
                make the tree's root-node pointer point to the new node;
            return;

if *nodepointer is a leaf node, say L,
    if L has space,                                            // usual case
    put entry on it, set newchildentry to null, and return;
    else,                                  // once in a while, the leaf is full
        split L: first d entries stay, rest move to brand new node L2;
        newchildentry = & ((smallest key value on L2, pointer to L2));
        set sibling pointers in L and L2;
        return;
endproc
```

Figure 10.1.0 Algorithrn for Insertion into B+ Tree of Order *d*

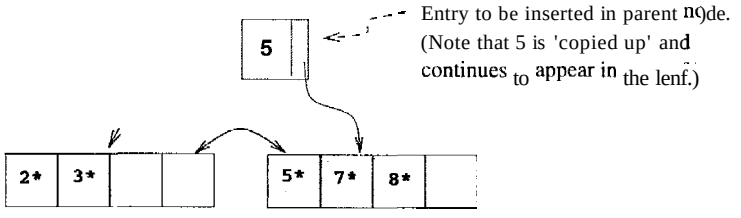


Figure 10.11 Split Leaf Pages during Insert of Entry 8*

The split pages in our example are shown in Figure 10.12. The index entry pointing to the new non-leaf node is the pair ⟨*17, pointer to new index-level page*⟩; note that the key value 17 is 'pushed up' the tree, in contrast to the splitting key value 5 in the leaf split, which was 'copied up.'

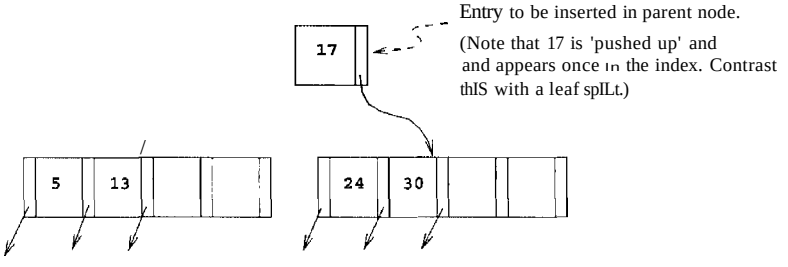


Figure 10.12 Split Index Pages during Insert of Entry 8*

The difference in handling leaf-level and index-level splits arises from the B+ tree requirement that all data entries $k*$ must reside in the leaves. This requirement prevents us from 'pushing up' 5 and leads to the slight redundancy of having some key values appearing in the leaf level as well as in some index level. However, range queries can be efficiently answered by just retrieving the sequence of leaf pages; the redundancy is a small price to pay for efficiency. In dealing with the index levels, we have more flexibility, and we 'push up' 17 to avoid having two copies of 17 in the index levels.

Now, since the split node was the old root, we need to create a new root node to hold the entry that distinguishes the two split index pages. The tree after completing the insertion of the entry 8* is shown in Figure 10.13.

One variation of the insert algorithm tries to redistribute entries of a node N with a sibling before splitting the node; this improves average occupancy. The sibling of a node N, in this context, is a node that is immediately to the left or right of N *and has the same parent as N.*

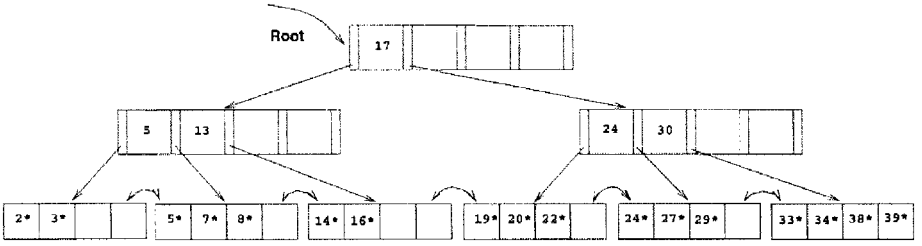


Figure 10.13 B+ Tree after Inserting Entry 8*

To illustrate redistribution, reconsider insertion of entry 8* into the tree shown in Figure 10.9. The entry belongs in the left-most leaf, which is full. However, the (only) sibling of this leaf node contains only two entries and can thus accommodate more entries. We can therefore handle the insertion of 8* with a redistribution. Note how the entry in the parent node that points to the second leaf has a new key value; we 'copy up' the new low key value on the second leaf. This process is illustrated in Figure 10.14.

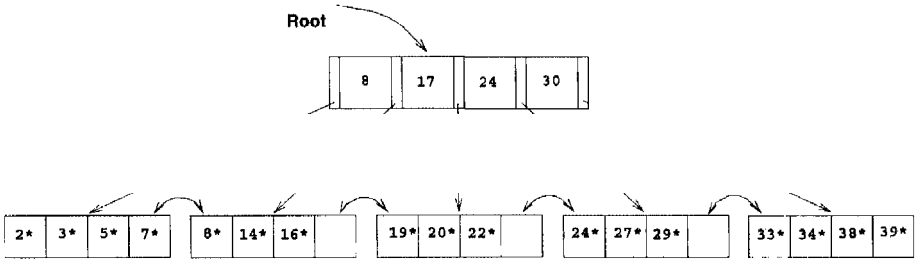


Figure 10.14 B+ Tree after Inserting Entry 8* Using Redistribution

To determine whether redistribution is possible, we have to retrieve the sibling. If the sibling happens to be full, we have to split the node anyway. On average, checking whether redistribution is possible increases I/O for index node splits, especially if we check both siblings. (Checking whether redistribution is possible may reduce I/O if the redistribution succeeds whereas a split propagates up the tree, but this case is very infrequent.) If the file is growing, average occupancy will probably not be affected much even if we do not redistribute. Taking these considerations into account, *not* redistributing entries at non-leaf levels usually pays off.

If a split occurs at the leaf level, however, we have to retrieve a neighbor to adjust the previous and next-neighbor pointers with respect to the newly created leaf node. Therefore, a limited form of redistribution makes sense: If a leaf node is full, fetch a neighbor node; if it has space and has the same parent,

redistribute the entries. Othenvise (the neighbor has different parent, Le., it is not a sibling, or it is also full) split the leaf node and a,djust the previous and next-neighbor pointers in the split node, the newly created neighbor, and the old neighbor.

10.6 DELETE

The algorithm for deletion takes an entry, finds the leaf node where it belongs, and deletes it. Pseudocode for the B+ tree deletion algorithm is given in Figure 10.15. The basic idea behind the algorithm is that we recursively delete the entry by calling the delete algorithm on the appropriate child node. We usually go down to the leaf node where the entry belongs, remove the entry from there, and return all the way back to the root node. Occasionally a node is at minimum occupancy before the deletion, and the deletion causes it to go below the occupancy threshold. When this happens, we must either redistribute entries from an adjacent sibling or merge the node with a sibling to maintain minimum occupancy. If entries are redistributed between two nodes, their parent node must be updated to reflect this; the key value in the index entry pointing to the second node must be changed to be the lowest search key in the second node. If two nodes are merged, their parent must be updated to reflect this by deleting the index entry for the second node; this index entry is pointed to by the pointer variable *oldchildentry* when the delete call returns to the parent node. If the last entry in the root node is deleted in this manner because one of its children was deleted, the height of the tree decreases by 1.

To illustrate deletion, let us consider the sample tree shown in Figure 10.13. To delete entry 19*, we simply remove it from the leaf page on which it appears, and we are done because the leaf still contains two entries. If we subsequently delete 20*, however, the leaf contains only one entry after the deletion. The (only) sibling of the leaf node that contained 20* has three entries, and we can therefore deal with the situation by redistribution; we move entry 24* to the leaf page that contained 20* and copy up the new splitting key (27, which is the new low key value of the leaf from which we borrowed 24*) into the parent. This process is illustrated in Figure 10.16.

Suppose that we now delete entry 24*. The affected leaf contains only one entry (22*) after the deletion, and the (only) sibling contains just two entries (27* and 29*). Therefore, we cannot redistribute entries. However, these two leaf nodes together contain only three entries and can be merged. \Vhile merging, we can 'toss' the entry (*27, pointer' to second leaf page)*) in the parent, which pointed to the second leaf page, because the second leaf page is elnpty after the merge and can be discarded. The right subtree of Figure 10.16 after this step in the deletion of entry 24* is shown in Figure 10.17.

```
proc delete (parentpointer, nodepointer, entry, oldchiIdentry)
// Deletes entry from s'ubtree with TOot '*nodepointer'; degree is d;
// 'oldchildentry' null initially, and null upon return unless child deleted
if *nodepointer is a non-leaf node, say N,
    find i such that K_i ≤ entry's key value < K_i+I;    // choose subtree
    delete(nodepointer, Pi, entry, oldchildentry);       // recursive delete
    if oldchildentry is null, return;          // usual case: child not deleted
    else,                                      // we discarded child node (see discussion)
        remove *oldchildentry from N,          // next, check for underflow
        if N has entries to spare,                                  // usual case
            set oldchildentry to null, return; // delete doesn't go further
        else,                                  // note difference wrt merging of leaf pages!
            get a sibling S of N:              // parentpointer arg used to find S
            if S has extra entries,
                redistribute evenly between Nand S through parent;
                set oldchildentry to null, return;
            else, merge Nand S                 // call node on rhs M
                oldchildentry = & (current entry in parent for M);
                pull splitting key from parent down into node on left;
                move all entries from M to node on left;
                discard empty node M, return;

if *nodepointer is a leaf node, say L,
    if L has entries to spare,                                  // usual case
        remove entry, set oldchildentry to null, and return;
    else,                                  // once in a while, the leaf becomes underfull
        get a sibling S of L;              // parentpointer used to find S
        if S has extra entries,
            redistribute evenly between Land S;
            find entry in parent for node on right;          // call it M
            replace key value in parent entry by new low-key value in M;
            set oldchildentry to null, return;
        else, merge Land S                 // call node on rhs M
            oldchildentry = & (current entry in parent for M);
            move all entries from M to node on left;
            discard empty node M, adjust sibling pointers, return;
endproc
```

Figure 10.15 Algorithm for Deletion from B+ Tree of Order d

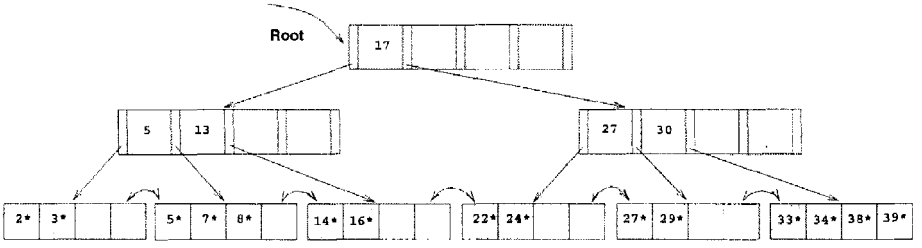


Figure 10.16 B+ Tree after Deleting Entries 19* and 20*

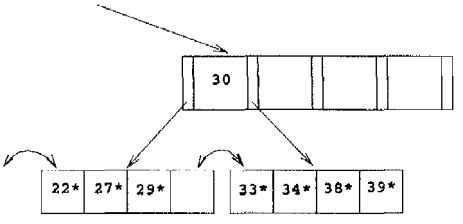


Figure 10.17 Partial B+ Tree during Deletion of Entry 24*

Deleting the entry (27, *pointer to second leaf page)* has created a non-Ieaf-Ievel page with just one entry, which is below the minimum of $d = 2$. To fix this problem, we must either redistribute or merge. In either case, we must fetch a sibling. The only sibling of this node contains just two entries (with key values 5 and 13), and so redistribution is not possible; we must therefore merge.

The situation when we have to merge two non-leaf nodes is exactly the opposite of the situation when we have to split a non-leaf node. We have to split a non-leaf node when it contains $2d$ keys and $2d + 1$ pointers, and we have to add another key--pointer pair. Since we resort to merging two non-leaf nodes only when we cannot redistribute entries between them, the two nodes must be minimally full; that is, each must contain d keys and $d + 1$ pointers prior to the deletion. After merging the two nodes and removing the key--pointer pair to be deleted, we have $2d - 1$ keys and $2d + 1$ pointers: Intuitively, the left-most pointer on the second merged node lacks a key value. To see what key value must be combined with this pointer to create a complete index entry, consider the parent of the two nodes being merged. The index entry pointing to one of the merged nodes must be deleted from the parent because the node is about to be discarded. The key value in this index entry is precisely the key value we need to complete the new merged node: The entries in the first node being merged, followed by the splitting key value that is 'pulled down' from the parent, followed by the entries in the second non-leaf node gives us a total of $2d$ keys and $2d + 1$ pointers, which is a full non-leaf node. Note how the splitting

key value in the parent is pulled down, in contrast to the case of merging two leaf nodes.

Consider the merging of two non-leaf nodes in our example. Together, the non-leaf node and the sibling to be merged contain only three entries, and they have a total of five pointers to leaf nodes. To merge the two nodes, we also need to pull down the index entry in their parent that currently discriminates between these nodes. This index entry has key value 17, and so we create a new entry *(17, left-most child pointer in sibling).* Now we have a total of four entries and five child pointers, which can fit on one page in a tree of order $d = 2$. Note that pulling down the splitting key 17 means that it will no longer appear in the parent node following the merge. After we merge the affected non-leaf node and its sibling by putting all the entries on one page and discarding the empty sibling page, the new node is the only child of the old root, which can therefore be discarded. The tree after completing all these steps in the deletion of entry 24* is shown in Figure 10.18.

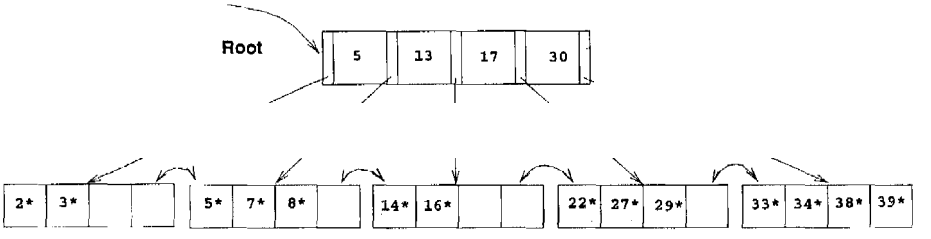


Figure 10.18 B+ Tree after Deleting Entry 24*

The previous examples illustrated redistribution of entries across leaves and merging of both leaf-level and non-leaf-level pages. The remaining case is that of redistribution of entries between non-leaf-level pages. To understand this case, consider the intermediate right subtree shown in Figure 10.17. We would arrive at the same intermediate right subtree if we try to delete 24* from a tree similar to the one shown in Figure 10.16 but with the left subtree and root key value as shown in Figure 10.19. The tree in Figure 10.19 illustrates an intermediate stage during the deletion of 24*. (Try to construct the initial tree.)

In contrast to the case when we deleted 24* from the tree of Figure 10.16, the non-leaf level node containing key value 30 now has a sibling that can spare entries (the entries with key values 17 and 20). We move these entries[3] over from the sibling. Note that, in doing so, we essentially push them through the

[3]It is sufficient to move over just the entry with key value 20, hut we are moving over two entries so illustrate what happens when several entries are redistributed.

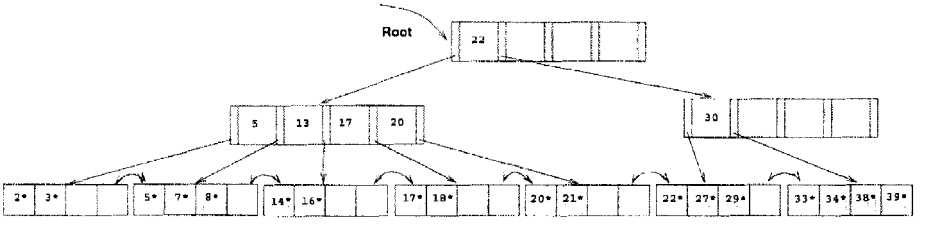


Figure 10.19 A B+ Tree during a Deletion

splitting entry in their parent node (the root), which takes care of the fact that 17 becomes the new low key value on the right and therefore must replace the old splitting key in the root (the key value 22). The tree with all these changes is shown in Figure 10.20.

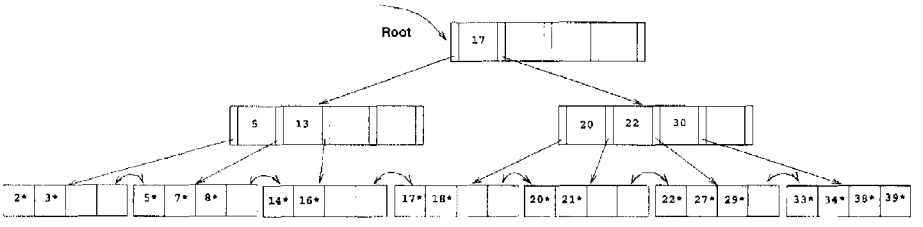


Figure 10.20 B+ Tree after Deletion

In concluding our discussion of deletion, we note that we retrieve only one sibling of a node. If this node has spare entries, we use redistribution; otherwise, we merge. If the node has a second sibling, it may be worth retrieving that sibling as well to check for the possibility of redistribution. Chances are high that redistribution is possible, and unlike merging, redistribution is guaranteed to propagate no further than the parent node. Also, the pages have more space on them, which reduces the likelihood of a split on subsequent insertions. (Remember, files typically grow, not shrink!) However, the number of times that this case arises (the node becomes less than half-full and the first sibling cannot spare an entry) is not very high, so it is not essential to implement this refinement of the basic algorithm that we presented.

10.7 DUPLICATES

The search, insertion, and deletion algorithms that we presented ignore the issue of duplicate keys, that is, several data entries with the same key value. We now discuss how duplica.tes can be handled.

Duplicate Handling in **Commercial** Systems: In a clustered index in Sybase ASE, the data rows are maintained in sorted order onthe page and in the eollection of data pages. The data pages are bidireetionally linked in sort order. Rows with duplicate keys are inserted into (or deleted from) the ordered set of rows. This may result in overflow pages of rows with duplieate keys being inserted into the page chain or empty overflow pages removed from the page chain. Insertion or deletion of a duplicate key does not affect the higher index levels unless a split or merge ofa non.-overflow page occurs. In IBM DB2, Oracle 8, and Microsoft SQL Server, duplicates are handled by adding a row id if necessary to eliminate duplicate key values.

The basic search algorithm assumes that all entries with a given key value reside on a single leaf page. One way to satisfy this assumption is to use *overflow pages* to deal with duplicates. (In ISAM, of course, we have overflow pages in any case, and duplicates are easily handled.)

Typically, however, we use an alternative approach for duplicates. We handle them just like any other entries and several leaf pages may contain entries with a given key value. To retrieve all data entries with a given key value, we must search for the *left-most* data entry with the given key value and then possibly retrieve more than one leaf page (using the leaf sequence pointers). Modifying the search algorithm to find the left-most data entry in an index with duplicates is an interesting exercise (in fact, it is Exercise 10.11).

One problem with this approach is that, when a record is deleted, if we use Alternative (2) for data entries, finding the corresponding data entry to delete in the B+ tree index could be inefficient because we may have to check several duplicate entries *(key, rid)* with the same *key* value. This problem can be addressed by considering the *rid* value in the data entry to be *part of the search key,* for purposes of positioning the data entry in the tree. This solution effectively turns the index into a *unique* index (i.e" no duplicates), Remember that a search key can be any sequence of fields in this variant, the rid of the data record is essentially treated as another field while constructing the search key.

Alternative (3) for data entries leads to a natural solution for duplicates, but if we have a large number of duplicates, a single data entry could span multiple pages. And of course, when a data record is deleted, finding the rid to delete from the corresponding data entry can be inefficient, The solution to this problem is similar to the one discussed previously for Alternative (2): We can

maintain the list of rids within each data entry in sorted order (say, by page number and then slot number if a rid consists of a page id and a slot id).

10.8 B+ TREES IN PRACTICE

In this section we discuss several important pragmatic issues.

10.8.1 Key Compression

The height of a B+ tree depends on the *number of data entries* and the *size of index entries.* The size of index entries determines the number of index entries that will fit on a page and, therefore, the *fan-out* of the tree. Since the height of the tree is proportional to *logfan-oud# of data entries),* and the number of disk l/Os to retrieve a data entry is equal to the height (unless some pages are found in the buffer pool), it is clearly important to maximize the fan-out to minimize the height.

An index entry contains a search key value and a page pointer. Hence the size depends primarily on the size of the search key value. If search key values are very long (for instance, the name Devarakonda Venkataramana Sathyanarayana Seshasayee Yellamanchali Murthy, or Donaudampfschifffahrts-kapitansanwiirtersmiitze), not many index entries will fit on a page: Fan-out is low, and the height of the tree is large.

On the other hand, search key values in index entries are used only to direct traffic to the appropriate leaf. When we want to locate data entries with a given search key value, we compare this search key value with the search key values of index entries (on a path from the root to the desired leaf). During the comparison at an index-level node, we want to identify two index entries with search key values kl and k_2 such that the desired search key value k falls between k_1 and $k2$. To accomplish this, we need not store search key values in their entirety in index entries.

For example, suppose we have two adjacent index entries in a node, with search key values 'David Smith' and 'Devarakonda ... ' To discriminate between these two values, it is sufficient to store the abbreviated forms 'Da' and 'De.' More generally, the lneaning of the entry 'David Smith' in the B+ tree is that every value in the subtree pointed to by the pointer to the left of 'David Smith' is less than 'David Smith,' and every value in the subtree pointed to by the pointer to the right of 'David Smith' is (greater than or equal to 'David Smith' and) less than 'Devarakonda ... '

B+ Trees in Real Systems: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support clustered and unclustered B+ tree indexes, with some differences in how they handle deletions and duplicate key values. In Sybase ASE, depending on the concurrency control schelne being used for the index, the deleted row is removed (with merging if the page occupancy goes below threshold) or simply marked as deleted; a garbage collection scheme is used to recover space in the latter case. In Oracle 8, deletions are handled by marking the row as deleted. To reclaim the space occupied by deleted records, we can rebuild the index online (i.e., while users continue to use the index) or coalesce underfull pages (which does not reduce tree height). Coalesce is in-place, rebuild creates a copy. Informix handles deletions by simply marking records as deleted. DB2 and SQL Server remove deleted records and merge pages when occupancy goes below threshold.

Oracle 8 also allows records from multiple relations to be co-clustered on the same page. The co-clustering can be based on a B+ tree search key or static hashing and up to 32 relations can be stored together.

To ensure such semantics for an entry is preserved, while compressing the entry with key 'David Smith,' we must examine the largest key value in the subtree to the left of 'David Smith' and the smallest key value in the subtree to the right of 'David Smith,' not just the index entries ('Daniel Lee' and 'Devarakonda ...') that are its neighbors. This point is illustrated in Figure 10.21; the value 'Davey Jones' is greater than 'Dav,' and thus, 'David Smith' can be abbreviated only to 'Davi,' not to 'Dav.'

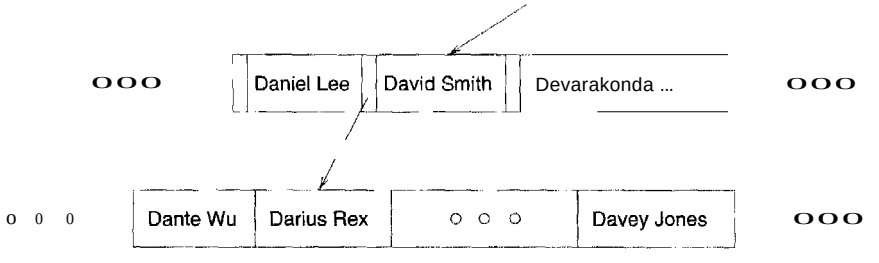


Figure 10.21 Example Illustrating Prefix Key Compression

This technique. called prefix key compression or simply key compression, is supported in many commercial implementations of B+ trees. It can substantially increase the fan-out of a tree. We do not discuss the details of the insertion and deletion algorithms in the presence of key compression.

10.8.2 Bulk-Loading a B+ Tree

Entries are added to a B+ tree in two ways. First, we may have an existing collection of data records with a B+ tree index on it; whenever a record is added to the collection, a corresponding entry must be added to the B+ tree as well. (Of course, a similar comment applies to deletions.) Second, we may have a collection of data records for which we want to create a B+ tree index on some key field(s). In this situation, we can start with an empty tree and insert an entry for each data record, one at a time, using the standard insertion algorithm. However, this approach is likely to be quite expensive because each entry requires us to start from the root and go down to the appropriate leaf page. Even though the index-level pages are likely to stay in the buffer pool between successive requests, the overhead is still considerable.

For this reason many systems provide a *bulk-loading* utility for creating a B+ tree index on an existing collection of data records. The first step is to sort the data entries k^* to be inserted into the (to be created) B+ tree according to the search key k. (If the entries are key-pointer pairs, sorting them does not mean sorting the data records that are pointed to, of course.) We use a running example to illustrate the bulk-loading algorithm. We assume that each data page can hold only two entries, and that each index page can hold two entries and an additional pointer (i.e., the B+ tree is assumed to be of order $d = 1$).

After the data entries have been sorted, we allocate an empty page to serve as the root and insert a pointer to the first page of (sorted) entries into it. We illustrate this process in Figure 10.22, using a sample set of nine sorted pages of data entries.

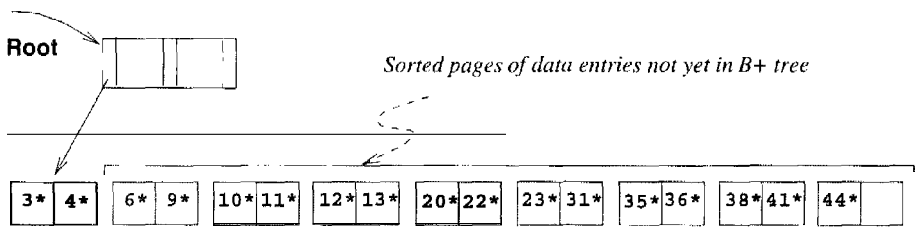


Figure 10.22 Initial Step in B+ Tree Bulk-Loading

We then add one entry to the root page for each page of the sorted data entries. The new entry consists of ⟨*low key value on page, pointer' to page)*. We proceed until the root page is full; see Figure 10.23.

To insert the entry for the next page of data entries, we must split the root and create a new root page. We show this step in Figure 10.24.

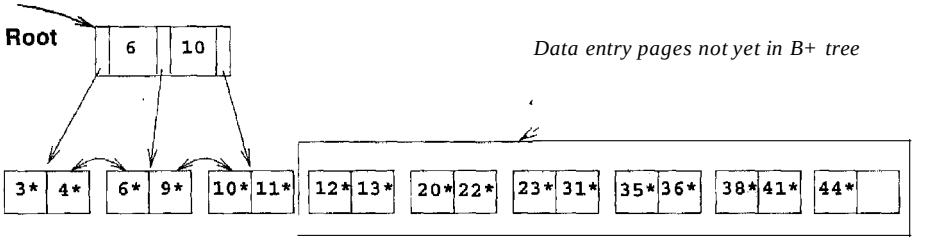


Figure 10.23 Root Page Fills up in B+ Tree Bulk-Loading

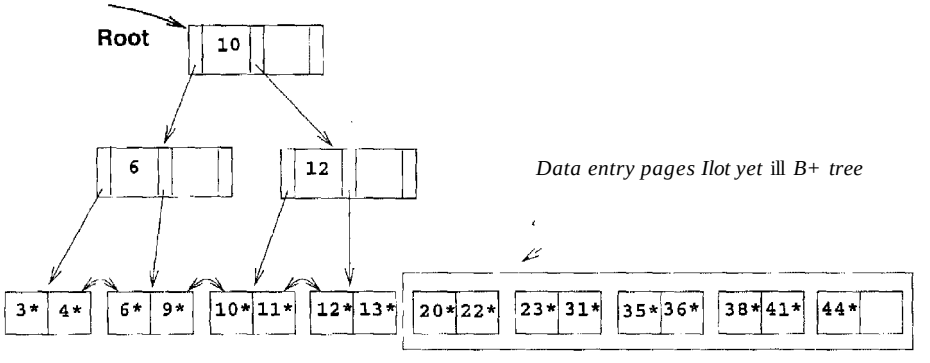


Figure 10.24 Page Split during B+ Tree Bulk-Loading

"We have redistributed the entries evenly between the two children of the root, in anticipation of the fact that the B+ tree is likely to grow. Although it is difficult (!) to illustrate these options when at most two entries fit on a page, we could also have just left all the entries on the old page or filled up some desired fraction of that page (say, 80 percent). These alternatives are simple variants of the basic idea.

To continue with the bulk-loading example, entries for the leaf pages are always inserted into the right-most index page just above the leaf level. 'When the right-most index page above the leaf level fills up, it is split. This action may cause a split of the right-most index page one step closer to the root, as illustrated in Figures 10.25 and 10.26.

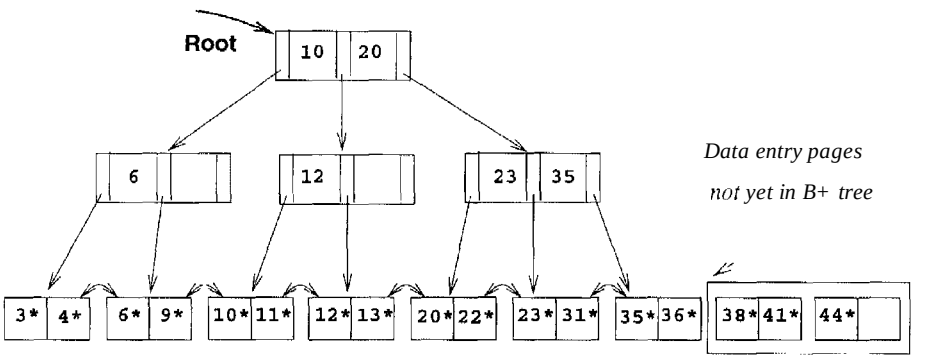


Figure 10.25 Before Adding Entry for Leaf Page Containing 38*

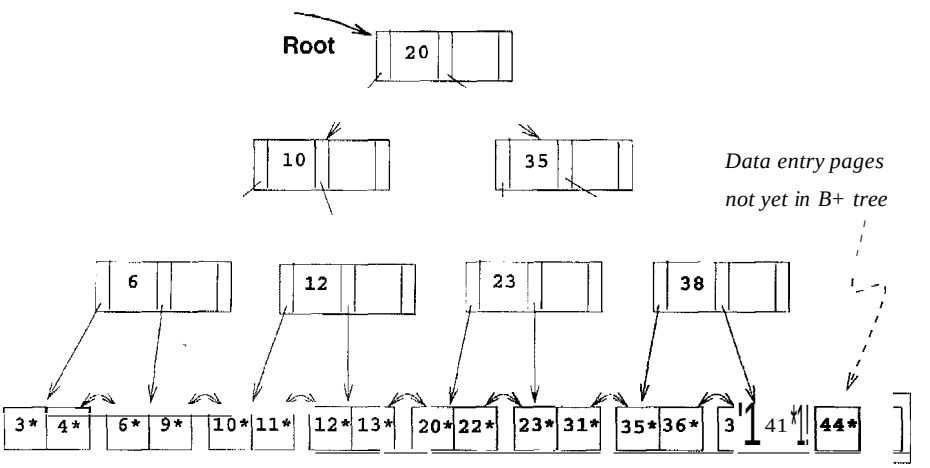


Figure 10.26 After Adding Entry for Leaf Page Containing 38*

Note that splits occur only on the right-most path from the root to the leaf level. We leave the completion of the bulk-loading example as a simple exercise.

Let us consider the cost of creating an index on an existing collection of records. This operation consists of three steps: (1) creating the data entries to insert in the index, (2) sorting the data entries, and (3) building the index from the sorted entries. The first step involves scanning the records and writing out the corresponding data entries; the cost is $(R + E)$ I/Os, where R is the number of pages containing records and E is the number of pages containing data entries. Sorting is discussed in Chapter 13; you will see that the index entries can be generated in sorted order at a cost of about $3E$ I/Os. These entries can then be inserted into the index as they are generated, using the bulk-loading algorithm discussed in this section. The cost of the third step, that is, inserting the entries into the index, is then just the cost of writing out all index pages.

10.8.3 The Order Concept

We presented B+ trees using the parameter *d* to denote minimum occupancy. It is worth noting that the concept of *order* (i.e., the parameter *d),* while useful for teaching B+ tree concepts, must usually be relaxed in practice and replaced by a physical space criterion; for example, that nodes must be kept at least half-full.

One reason for this is that leaf nodes and non-leaf nodes can usually hold different numbers of entries. Recall that B+ tree nodes are disk pages and non-leaf nodes contain only search keys and node pointers, while leaf nodes can contain the actual data records. Obviously, the size of a data record is likely to be quite a bit larger than the size of a search entry, so many more search entries than records fit on a disk page.

A second reason for relaxing the order concept is that the search key may contain a character string field (e.g., the *name* field of Students) whose size varies from record to record; such a search key leads to variable-size data entries and index entries, and the number of entries that will fit on a disk page becomes variable.

Finally, even if the index is built on a fixed-size field, several records may still have the same search key value (e.g., several Students records may have the same *gpa* or *name* value). This situation can also lead to variable-size leaf entries (if we use Alternative (3) for data entries). Because of all these complications, the concept of order is typically replaced by a simple physical criterion (e.g., merge if possible when more than half of the space in the node is unused).

10.8.4 The Effect of Inserts and Deletes on Rids

If the leaf pages contain data records-that is, the B+ tree is a clustered index-then operations such as splits, merges, and redistributions can change rids. Recall that a typical representation for a rid is some combination of (physical) page number and slot number. This scheme allows us to move records within a page if an appropriate page format is chosen but not across pages, as is the case with operations such as splits. So unless rids are chosen to be independent of page numbers, an operation such as split or merge in a clustered B+ tree may require compensating updates to other indexes on the same data.

A similar comment holds for any dynamic clustered index, regardless of whether it is tree-based or hash-based. Of course, the problem does not arise with nonclustered indexes, because only index entries are moved around.

10.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Why are tree-structured indexes good for searches, especially range selections? (Section 10.1)
- Describe how search, insert, and delete operations work in ISAM indexes. Discuss the need for overflow pages, and their potential impact on performance. What kinds of update workloads are ISAM indexes most vulnerable to, and what kinds of workloads do they handle well? (Section 10.2)
- Only leaf pages are affected in updates in ISAM indexes. Discuss the implications for locking and concurrent access. Compare ISAM and B+ trees in this regard. (Section 10.2.1)
- What are the main differences between ISAM and B+ tree indexes? (Section 10.3)
- What is the *order* of a B+ tree? Describe the format of nodes in a B+ tree. Why are nodes at the leaf level linked? (Section 10.3)
- How rnany nodes must be examined for equality search in a B+ tree? How many for a range selection? Compare this with ISAM. (Section 10.4)
- Describe the B+ tree insertion algorithm, and explain how it eliminates overflow pages. Under what conditions can an insert increase the height of the tree? (Section 10.5)
- During deletion, a node might go below the minimum occupancy threshold. How is this handled? Under what conditions could a deletion decrease the height of the tree? (Section 10.6)

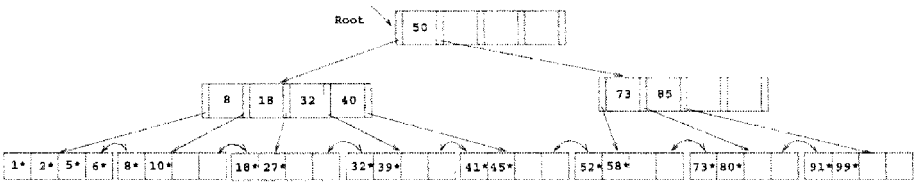


Figure 10.27 Tree for Exercise 10.1

- Why do duplicate search keys require modifications to the implementation of the basic B+ tree operations? (Section 10.7)
- \What is *key compression,* and why is it important? (Section 10.8.1)
- How can a new B+ tree index be efficiently constructed for a set of records? Describe the *bulk-loading* algorithm. (Section 10.8.2)
- Discuss the impact of splits in clustered B+ tree indexes. (Section 10.8.4)

EXERCISES

Exercise 10.1 Consider the B+ tree index of order d = 2 shown in Figure 10.27.

1. Show the tree that would result from inserting a data entry with key 9 into this tree.
2. Show the B+ tree that would result from inserting a data entry with key 3 into the original tree. How many page reads and page writes does the insertion require?
3. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the left sibling is checked for possible redistribution.
4. Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the right sibling is checked for possible redistribution.
5. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 46 and then deleting the data entry with key 52.
6. Show the B+ tree that would result from deleting the data entry with key 91 from the original tree.
7. Show the B+ tree that would result from starting with the original tree, inserting a data entry with key 59, and then deleting the data entry with key 91.
8. Show the B+ tree that \would result from successively deleting the data entries with keys 32, 39, 41, 45, and 73 from the original tree.

Exercise 10.2 Consider the B+ tree index shown in Figure 10.28, which uses Alternative (1) for data entries. Each intermediate node can hold up to five pointers and four key values. Each leaf can hold up to four records, and leaf nodes are doubly linked as usual, although these links are not shown in the figure. Answer the following questions.

1. Name all the tree nodes that mllst be fetched to answer the following query: "Get all records with search key greater than 38."

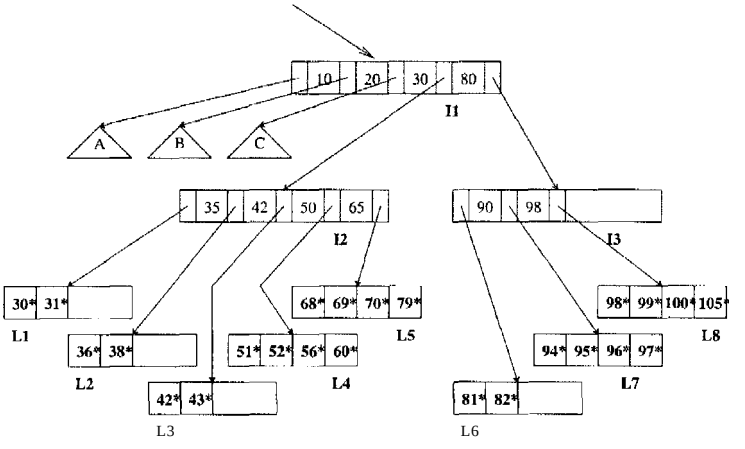


Figure 10.28 Tree for Exercise 10.2

2. Insert a record with search key 109 into the tree.
3. Delete the record with search key 81 from the (original) tree.
4. Name a search key value such that inserting it into the (original) tree would cause an increase in the height of the tree.
5. Note that subtrees A, B, and C are not fully specified. Nonetheless, what can you infer about the contents and the shape of these trees?
6. How would your answers to the preceding questions change if this were an ISAM index?
7. Suppose that this is an ISAM index. What is the minimum number of insertions needed to create a chain of three overflow pages?

Exercise 10.3 Answer the following questions:

1. What is the minimum space utilization for a B+ tree index?
2. What is the minimum space utilization for an ISAM index?
3. If your database system supported both a static and a dynamic tree index (say, ISAM and B+ trees), would you ever consider using the *static* index in preference to the *dynamic* index?

Exercise 10.4 Suppose that a page can contain at most four data values and that all data values are integers. Using only B+ trees of order 2, give examples of each of the following:

1. A B+ tree whose height changes from 2 to 3 when the value 25 is inserted. Show your structure before and after the insertion.
2. A B+ tree in which the deletion of the value 25 leads to a redistribution. Show your structure before and after the deletion.
3. A B+ tree in which the deletion of the value 25 causes a merge of two nodes but without altering the height of the tree.
4. An ISAM structure with four buckets, none of which has an overflow page. Further, every bucket has space for exactly one more entry. Show your structure before and after inserting two additional values, chosen so that an overflow page is created.

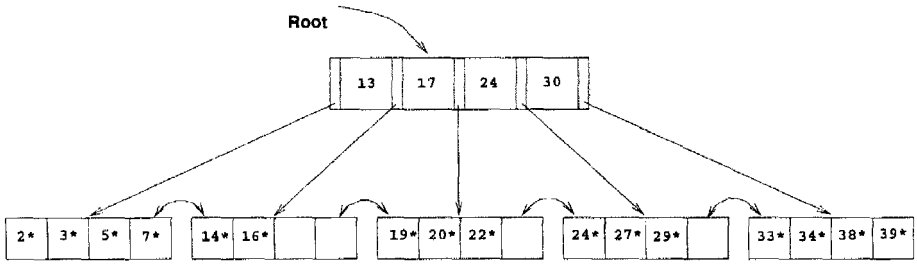


Figure 10.29 Tree for Exercise 10.5

Exercise 10.5 Consider the B+ tree shown in Figure 10.29.

1. Identify a list of five data entries such that:
   (a) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert *a*, insert *b*, delete *b*, delete *a)* results in the original tree.
   (b) Inserting the entries in the order shown and then deleting them in the opposite order (e.g., insert *a*, insert *b*, delete *b*, delete *a)* results in a different tree.
2. What is the minimum number of insertions of data entries with distinct keys that will cause the height of the (original) tree to change from its current value (of 1) to 3?
3. Would the minimum number of insertions that will cause the original tree to increase to height 3 change if you were allowed to insert duplicates (multiple data entries with the same key), assuming that overflow pages are not used for handling duplicates?

Exercise 10.6 Answer Exercise 10.5 assuming that the tree is an ISAM tree! (Some of the examples asked for may not exist-if so, explain briefly.)

Exercise 10.7 Suppose that you have a sorted file and want to construct a dense primary B+ tree index on this file.

1. One way to accomplish this task is to scan the file, record by record, inserting each one using the B+ tree insertion procedure. What performance and storage utilization problems are there with this approach?
2. Explain how the bulk-loading algorithm described in the text improves upon this scheme.

Exercise 10.8 Assume that you have just built a dense B+ tree index using Alternative (2) on a heap file containing 20,000 records. The key field for this B+ tree index is a 40-byte string, and it is a candidate key. Pointers (Le., record ids and page ids) are (at most) 10-byte values. The size of one disk page is 1000 bytes. The index was built in a bottom-up fashion using the bulk-loading algorithm, and the nodes at each level were filled up as much as possible.

1. Ho\v many levels does the resulting tree have?
2. For each level of the trec, how many nodes are at that level?
3. How many levels would the resulting tree have if key compression is llsed and it reduces the average size of each key in an entry to 10 bytes?

sid	*name*	*login*	*age*	*gpa*
53831	Maclayall	maclayan@music	11	1.8
53832	Guldu	guldu@music	12	3.8
53666	Jones	jones@cs	18	3.4
53901	Jones	jones@toy	18	3.4
53902	Jones	jones@physics	18	3.4
53903	Jones	jones@english	18	3.4
53904	Jones	jones(ggenetics	18	3.4
53905	Jones	jones@astro	18	3.4
53906	Jones	jones@chem	18	3.4
53902	Jones	jones@sanitation	18	3.8
53688	Smith	smith@ee	19	3.2
53650	Smith	smith@math	19	3.8
54001	Smith	smith@ee	19	3.5
54005	Smith	smith@cs	19	3.8
54009	Smith	smith@astro	19	2.2

Figure 10.30 An Instance of the Students Relation

4. How many levels would the resulting tree have without key compression but with all pages 70 percent full?

Exercise 10.9 The algorithms for insertion and deletion into a B+ tree are presented as recursive algorithms. In the code for *insert*, for instance, a call is made at the parent of a node *N* to insert into (the subtree rooted at) node *N*, and when this call returns, the current node is the parent of *N*. Thus, we do not maintain any 'parent pointers' in nodes of B+ tree. Such pointers are not part of the B+ tree structure for a good reason, as this exercise demonstrates. An alternative approach that uses parent pointers--again, remember that such pointers are *not* part of the standard B+ tree structure!-in each node appears to be simpler:

> Search to the appropriate leaf using the search algorithm; then insert the entry and split if necessary, with splits propagated to parents if necessary (using the parent pointers to find the parents).

Consider this (unsatisfactory) alternative approach:

I. Suppose that an internal node *N* is split into nodes *N* and *N2*. What can you say about the parent pointers in the children of the original node *N?*

2. Suggest two ways of dealing with the inconsistent parent pointers in the children of node *N*.

3. For each of these suggestions, identify a potential (major) disadvantage.

4. What conclusions can you draw from this exercise?

Exercise 10.10 Consider the instance of the Students relation shown in Figure 10.30. Show a B+ tree of order 2 in each of these cases, assuming that duplicates are handled using overflow pages. Clearly indicate what the data entries are (i.e., do not use the *k** convention).

1. A B+ tree index on *age* using Alternative (1) for data entries.
2. A dense B+ tree index on *gpa* using Alternative (2) for data entries. For this question, assume that these tuples are stored in a sorted file in the order shown in the figure: The first tuple is in page 1, slot 1; the second tuple is in page 1, slot 2; and so on. Each page can store up to three data records. You can use *⟨page-id, slot⟩* to identify a tuple.

Exercise 10.11 Suppose that duplicates are handled using the approach without overflow pages discussed in Section 10.7. Describe an algorithm to search for the left-most occurrence of a data entry with search key value K.

Exercise 10.12 Answer Exercise 10.10 assuming that duplicates are handled without using overflow pages, using the alternative approach suggested in Section 9.7.

PROJECT-BASED EXERCISES

Exercise 10.13 Compare the public interfaces for heap files, B+ tree indexes, and linear hashed indexes. What are the similarities and differences? Explain why these similarities and differences exist.

Exercise 10.14 This exercise involves using Minibase to explore the earlier (non-project) exercises further.

1. Create the trees shown in earlier exercises and visualize them using the B+ tree visualizer in Minibase.
2. Verify your answers to exercises that require insertion and deletion of data entries by doing the insertions and deletions in Minibase and looking at the resulting trees using the visualizer.

Exercise 10.15 *(Note to instructors: Additional details must be provided if this exercise* is *assigned; see Appendix 30.)* Implement B+ trees on top of the lower-level code in Minibase.

BIBLIOGRAPHIC NOTES

The original version of the B+ tree was presented by Bayer and McCreight [69]. The B+ tree is described in [442] and [194]. B tree indexes for skewed data distributions are studied in [260]. The VSAM indexing structure is described in [764]. Various tree structures for supporting range queries are surveyed in [79]. An early paper on multiattribute search keys is [498].

References for concurrent access to B+ trees are in the bibliography for Chapter 17.

11

HASH-BASED INDEXING

- What is the intuition behind hash-structured indexes? Why are they especially good for equality searches but useless for range selections?
- What is Extendible Hashing? How does it handle search, insert, and delete?
- What is Linear Hashing? How does it handle search, insert, and delete?
- What are the similarities and differences between Extendible and Linear Hashing?
- **Key concepts:** hash function, bucket, primary and overflow pages, static versus dynamic hash indexes; Extendible Hashing, directory of buckets, splitting a bucket, global and local depth, directory doubling, collisions and overflow pages; Linear Hashing, rounds of splitting, family of hash functions, overflow pages, choice of bucket to split and time to split; relationship between Extendible Hashing's directory and Linear Hashing's family of hash functiolis, need for overflow pages in both schemes in practice, use of a directory for Linear Hashing.

J

Not chaos-like, together crushed and bruised,
But, as the world harmoniously confused:
Where order in variety we see.

Alexander Pope, *Windsor Forest*

In this chapter we consider file organizations that are excellent for equality selections. The basic idea is to use a *hashing function,* which maps values

in a search field into a range of *b'ucket numbers* to find the page on which a desired data entry belongs. We use a simple scheme called *Static Hashing* to introduce the idea. This scheme, like ISAM, suffers from the problem of long overflow chains, which can affect performance. Two solutions to the problem are presented. The *Extendible Hashing* scheme uses a directory to support inserts and deletes efficiently with no overflow pages. The *Linear Hashing* scheme uses a clever policy for creating new buckets and supports inserts and deletes efficiently without the use of a directory. Although overflow pages are used, the length of overflow chains is rarely more than two.

Hash-based indexing techniques cannot support range searches, unfortunately. n'ee-based indexing techniques, discussed in Chapter 10, can support range searches efficiently and are almost as good as hash-based indexing for equality selections. Thus, many commercial systems choose to support only tree-based indexes. Nonetheless, hashing techniques prove to be very useful in implementing relational operations such as joins, as we will see in Chapter 14. In particular, the Index Nested Loops join method generates many equality selection queries, and the difference in cost between a hash-based index and a tree-based index can become significant in this context.

The rest of this chapter is organized as follows. Section 11.1 presents Static Hashing. Like ISAM, its drawback is that performance degrades as the data grows and shrinks. We discuss a dynamic hashing technique, called *Extendible Hashing,* in Section 11.2 and another dynamic technique, called *Linear Hashing,* in Section 11.3. We compare Extendible and Linear Hashing in Section 11.4.

11.1 STATIC HASHING

The Static Hashing scheme is illustrated in Figure 11.1. The pages containing the data can be viewed as a collection of buckets, with one primary page and possibly additional overflow pages per bucket. A file consists of buckets a through $N - 1$, with one primary page per bucket initially. Buckets contain *data entTies,* which can be any of the three alternatives discussed in Chapter 8.

To search for a data entry, we apply a hash function h to identify the bucket to which it belongs and then search this bucket. To speed the search of a bucket, we can maintain data entries in sorted order by search key value; in this chapter, we do not sort entries, and the order of entries within a bucket has no significance. To insert a data entry, we use the hash function to identify the correct bucket and then put the data entry there. If there is no space for this data entry, we allocate a new *overflow* page, put the data entry on this page, and add the page to the overflow chain of the bucket. To delete a data

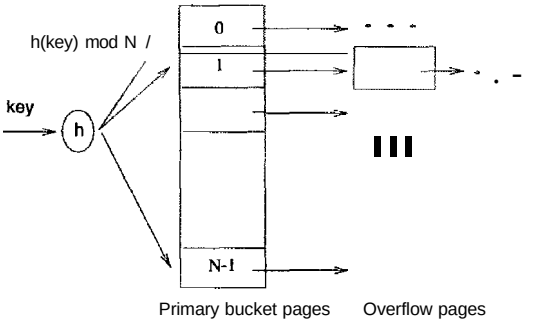


Figure 11.1 Static Hashing

entry, we use the hashing function to identify the correct bucket, locate the data entry by searching the bucket, and then remove it. If this data entry is the last in an overflow page, the overflow page is removed from the overflow chain of the bucket and added to a list of *free pages.*

The hash function is an important component of the hashing approach. It must distribute values in the domain of the search field uniformly over the collection of buckets. If we have N buckets, numbered 0 through $N - 1$, a hash function h of the form $h(value) = (a * value + b)$ works well in practice. (The bucket identified is $h(value) \bmod N$.) The constants a and b can be chosen to 'tune' the hash function.

Since the number of buckets in a Static Hashing file is known when the file is created, the primary pages can be stored on successive disk pages. Hence, a search ideally requires just one disk I/O, and insert and delete operations require two I/Os (read and write the page), although the cost could be higher in the presence of overflow pages. As the file grows, long overflow chains can develop. Since searching a bucket requires us to search (in general) all pages in its overflow chain, it is easy to see how performance can deteriorate. By initially keeping pages 80 percent full, we can avoid overflow pages if the file does not grow too Illuch, but in general the only way to get rid of overflow chains is to create a new file with more buckets.

The main problem with Static Hashing is that the number of buckets is fixed. If a file shrinks greatly, a lot of space is wasted; more important, if a file grows a lot, long overflow chains develop, resulting in poor performance. Therefore, Static Hashing can be compared to the ISAM structure (Section 10.2), which can also develop long overflow chains in case of insertions to the same leaf. Static Hashing also has the same advantages as ISAM with respect to concurrent access (see Section 10.2.1).

One simple alternative to Static Hashing is to periodically 'rehash' the file to restore the ideal situation (no overflow chains, about 80 percent occupancy). However, rehashing takes time and the index cannot be used while rehashing is in progress. Another alternative is to use **dynamic hashing** techniques such as Extendible and Linear Hashing, which deal with inserts and deletes gracefully. We consider these techniques in the rest of this chapter.

11.1.1 Notation and Conventions

In the rest of this chapter, we use the following conventions. As in the previous chapter, record with search key *k,* we denote the index data entry by *k**. For hash-based indexes, the first step in searching for, inserting, or deleting a data entry with search key *k* is to apply a hash function *h* to *k;* we denote this operation by *h(k),* and the value *h(k)* identifies the bucket for the data entry $k*$. Note that two different search keys can have the same hash value.

11.2 EXTENDIBLE HASHING

To understand Extendible Hashing, let us begin by considering a Static Hashing file. If we have to insert a new data entry into a full bucket, we need to add an overflow page. If we do not want to add overflow pages, one solution is to reorganize the file at this point by doubling the number of buckets and redistributing the entries across the new set of buckets. This solution suffers from one major defect--the entire file has to be read, and twice as many pages have to be written to achieve the reorganization. This problem, however, can be overcome by a simple idea: Use a **directory** of pointers to buckets, and double the size of the number of buckets by doubling just the directory and splitting *only* the bucket that overflowed.

To understand the idea, consider the sample file shown in Figure 11.2. The directory consists of an array of size 4, with each element being a pointer to a bucket.. (The *global depth* and *local depth* fields are discussed shortly, ignore them for now.) To locate a data entry, we apply a hash function to the search field and take the last 2 bits of its binary representation to get a number between 0 and 3. The pointer in this array position gives us the desired bucket; we assume that each bucket can hold four data entries. Therefore, to locate a data entry with hash value 5 (binary 101), we look at directory element 01 and follow the pointer to the data page (bucket B in the figure).

To insert a data entry, we search to find the appropriate bucket.. For example, to insert a data entry with hash value 13 (denoted as 13*), we examine directory element 01 and go to the page containing data entries 1*, 5*, and 21*. Since

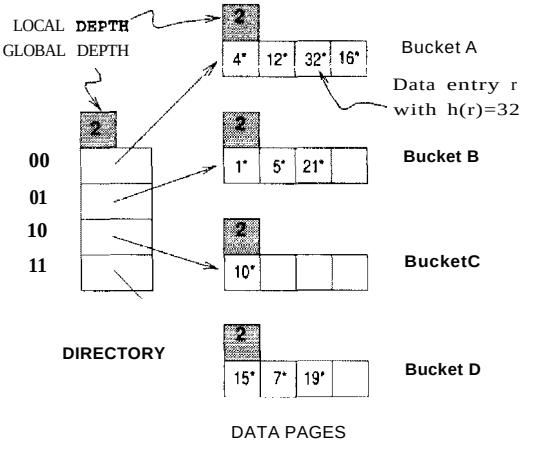


Figure 11.2 Example of an Extendible Hashed File

the page has space for an additional data entry, we are done after we insert the entry (Figure 11.3).

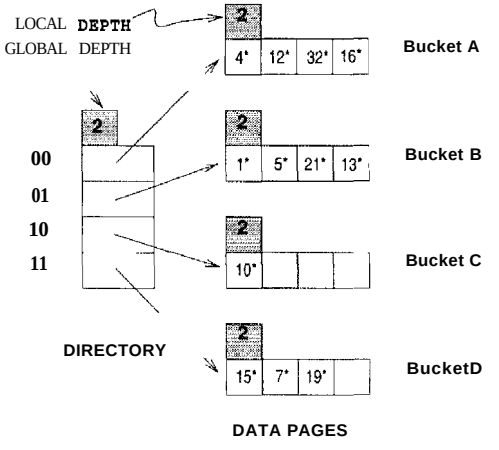


Figure 11.3 After Inserting Entry r with $h(r) = 13$

Next, let us consider insertion of a data entry into a full bucket. The essence of the Extcndible Hashing idea lies in how we deal with this case. Consider the insertion of data entry 20* (binary 10100). Looking at directory clement 00, we arc led to bucket A, which is already full. We 111Ust first **split** the bucket

by allocating a new bucket[1] and redistributing the contents (including the new entry to be inserted) across the old bucket and its 'split image.' To redistribute entries across the old bucket and its split image, we consider the last *three* bits of *h(T);* the last two bits are 00, indicating a data entry that belongs to one of these two buckets, and the third bit discriminates between these buckets. The redistribution of entries is illustrated in Figure 11.4.

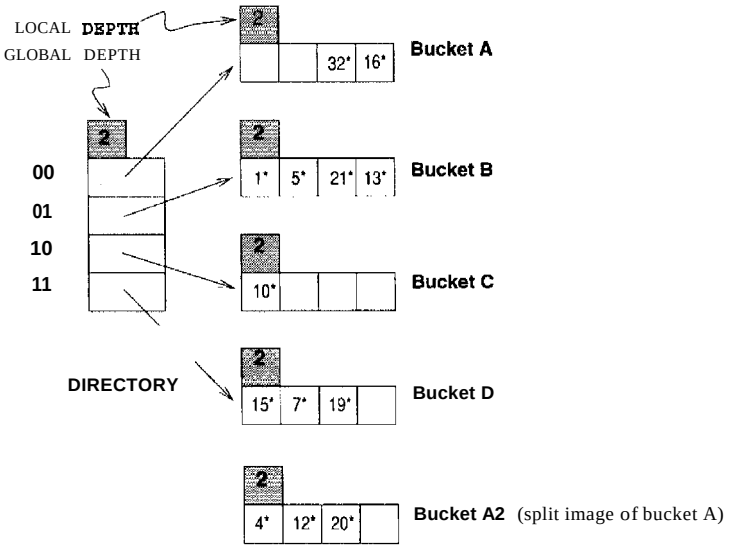


Figure 11.4 While Inserting Entry r with *h(r}=20*

Note a problem that we must now resolve—we need three bits to discriminate between two of our data pages (A and A2), but the directory has only enough slots to store all two-bit patterns. The solution is to *double the directory.* Elements that differ only in the third bit from the end are said to 'correspond': *COT-r'esponding elements* of the directory point to the same bucket with the exception of the elements corresponding to the split bucket. In our example, bucket a was split; so, new directory element 000 points to one of the split versions and new element 100 points to the other. The sample file after completing all steps in the insertion of 20* is shown in Figure 11.5.

Therefore, doubling the file requires allocating a new bucket page, writing both this page and the old bucket page that is being split, and doubling the directory array. The directory is likely to be much smaller than the file itself because each element is just a page-id, and can be doubled by simply copying it over

[1]Since there are 'no overflow pages in Extendible Hashing, a bucket can be thought of as a single page.

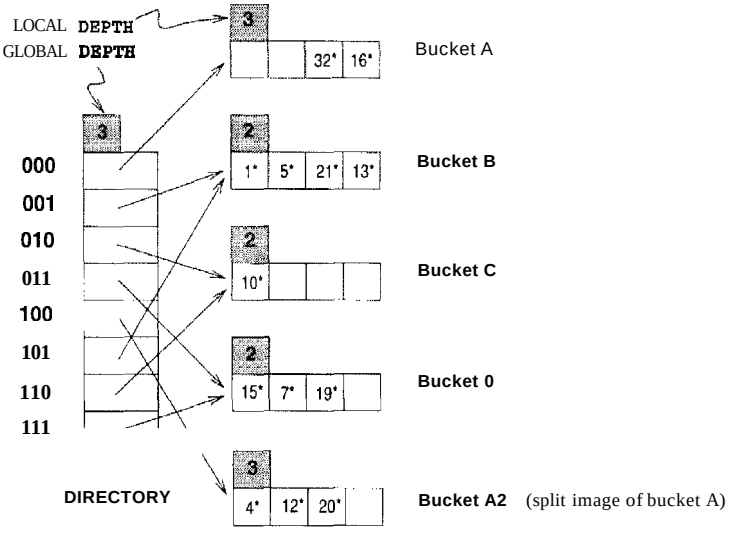


Figure 11.5 After Inserting Entry r with $h(r) = 20$

(and adjusting the elements for the split buckets). The cost of doubling is now quite acceptable.

We observe that the basic technique used in Extendible Hashing is to treat the result of applying a hash function h as a binary number and interpret the last d bits, where d depends on the size of the directory, as an offset into the directory. In our example, d is originally 2 because we only have four buckets; after the split, d becomes 3 because we now have eight buckets. A corollary is that, when distributing entries across a bucket and its split image, we should do so on the basis of the dth bit. (Note how entries are redistributed in our example; see Figure 11.5.) The number d, called the **global depth** of the hashed file, is kept as part of the header of the file. It is used every time we need to locate a data entry.

An important point that arises is whether splitting a bucket necessitates a directory doubling. Consider our example, as shown in Figure 11.5. If we now insert 9*, it belongs in bucket B; this bucket is already full. We can deal with this situation by splitting the bucket and using directory elements 001 and 10] to point to the bucket and its split image, as shown in Figure 11.6.

Hence, a bucket split does not necessarily require a directory doubling. However, if either bucket A or A2 grows full and an insert then forces a bucket split, we are forced to double the directory again.

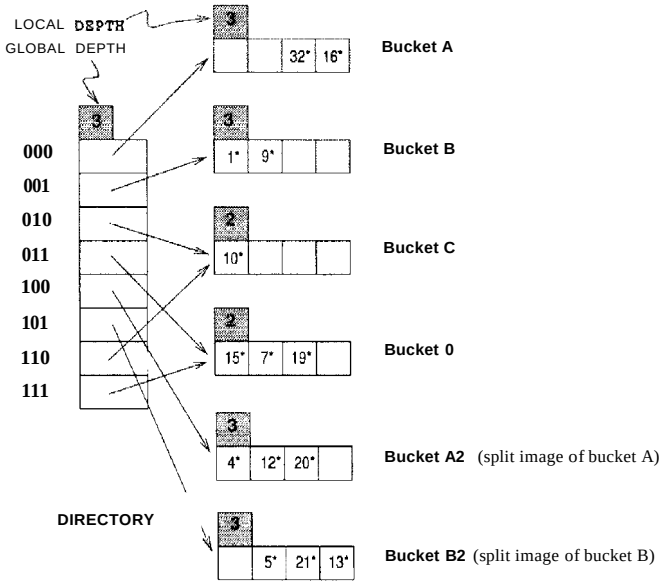


Figure 11.6 After Inserting Entry r with $h(r) = 9$

To differentiate between these cases and determine whether a directory doubling is needed, we maintain a **local depth** for each bucket. If a bucket whose local depth is equal to the global depth is split, the directory must be doubled. Going back to the example, when we inserted 9* into the index shown in Figure 11.5, it belonged to bucket B with local depth 2, whereas the global depth was 3. Even though the bucket was split, the directory did not have to be doubled. Buckets A and A2, on the other hand, have local depth equal to the global depth, and, if they grow full and are split, the directory must then be doubled.

Initially, all local depths are equal to the global depth (which is the number of bits needed to express the total number of buckets). We increment the global depth by 1 each time the directory doubles, of course. Also, whenever a bucket is split (whether or not the split leads to a directory doubling), we increment by 1 the local depth of the split bucket and assign this same (incremented) local depth to its (newly created) split image. Intuitively, if a bucket has local depth l, the hash values of data entries in it agree on the last l bits; further, no data entry in any other bucket of the file has a hash value with the same last l bits. A total of 2^{d-l} directory elernents point to a bucket with local depth l; if $d = l$, exactly one directory element points to the bucket and splitting such a bucket requires directory doubling.

A final point to note is that we can also use the first *d* bits (the *most significant* bits) instead of the last *d (least significant* bits), but in practice the *last d* bits are used. The reason is that a directory can then be doubled simply by copying it.

In summary, a data entry can be located by computing its hash value, taking the last *d* bits, and looking in the bucket pointed to by this directory element. For inserts, the data entry is placed in the bucket to which it belongs and the bucket is split if necessary to make space. A bucket split leads to an increase in the local depth and, if the local depth becomes greater than the global depth as a result, to a directory doubling (and an increase in the global depth) as well.

For deletes, the data entry is located and removed. If the delete leaves the bucket empty, it can be merged with its split image, although this step is often omitted in practice. Merging buckets decreases the local depth. If each directory element points to the same bucket as its split image (i.e., 0 and 2^{d-1} point to the same bucket, namely, A; 1 and $2^{d-1} + 1$ point to the same bucket, namely, B, which mayor may not be identical to A; etc.), we can halve the directory and reduce the global depth, although this step is not necessary for correctness.

The insertion examples can be worked out backwards as examples of deletion. (Start with the structure shown after an insertion and delete the inserted element. In each case the original structure should be the result.)

If the directory fits in memory, an equality selection can be answered in a single disk access, as for Static Hashing (in the absence of overflow pages), but otherwise, two disk I/Os are needed. As a typical example, a 100MB file with 100 bytes per data entry and a page size of 4KB contains 1 million data entries and only about 25,000 elements in the directory. (Each page/bucket contains roughly 40 data entries, and we have one directory element per bucket.) Thus, although equality selections can be twice as slow as for Static Hashing files, chances are high that the directory will fit in memory and performance is the same as for Static Hashing files.

On the other hand, the directory grows in spurts and can become large for *skewed data distributions* (where our assumption that data pages contain roughly equal numbers of data entries is not valid). In the context of hashed files, in a skewed data distribution the distribution of *hash values of search field values* (rather than the distribution of search field values themselves) is skewed (very 'bursty' or nonuniform). Even if the distribution of search values is skewed, the choice of a good hashing function typically yields a fairly uniform distribution of hash va.lues; skew is therefore not a problem in practice.

F\lrther, collisions, or data entries with the same hash value, cause a problem and must be handled specially: \Vhen more data entries than \vill fit on a page have the same hash value, we need overflow pages.

11.3 LINEAR HASHING

Linear Hashing is a dynamic hashing technique, like Extendible Hashing, adjusting gracefully to inserts and deletes. In contrast to Extendible Hashing, it does not require a directory, deals naturally with collisions, and offers a lot of flexibility with respect to the timing of bucket splits (allowing us to trade off slightly greater overflow chains for higher average space utilization). If the data distribution is very skewed, however, overflow chains could cause Linear Hashing performance to be worse than that of Extendible Hashing.

The scheme utilizes a *family* of hash functions h_a, hI, *h2,* ..., with the property that each function's range is twice that of its predecessor. That is, if *hi* maps a data entry into one of M buckets, h_i+I maps a data entry into one of $2M$ buckets. Such a family is typically obtained by choosing a hash function hand an initial number N ofbuckets,[2] and defining *hi(value) = h(value) mod* $(2^i N)$. If N is chosen to be a power of 2, then we apply h and look at the last d_i bits; d_o is the number of bits needed to represent N, and $d_i = d_a + i$. Typically we choose h to be a function that maps a data entry to some integer. Suppose that we set the initial number N of buckets to be 32. In this case d_o is 5, and h_a is therefore $h \bmod 32$, that is, a number in the range 0 to 31. The value of d_1 is $d_o + 1 = 6$, and hI is $h \bmod (2 * 32)$, that is, a number in the range 0 to 63. Then h_2 yields a number in the range 0 to 127, and so oII.

The idea is best understood in terms of **rounds** of splitting. During round number *Level,* only hash functions h_{Level} and $h_{Level+1}$ are in use. The buckets in the file at the beginning of the round are split, one by one from the first to the last bucket, thereby doubling the number of buckets. At any given point within a round, therefore, we have buckets that have been split, buckets that are yet to be split, and buckets created by splits in this round, as illustrated in Figure 11.7.

Consider how we search for a data entry with a given search key value. \Ve apply hash function h_{Level}, and if this leads us to one of the unsplit buckets, we simply look there. If it leads us to one of the split buckets, the entry may be there or it may have been moved to the new bucket created earlier in this round by splitting this bucket; to determine which of the two buckets contains the entry, we apply $h_{Level+1}$.

[2] Note that 0 to IV - 1 is *not* the range of fl.!

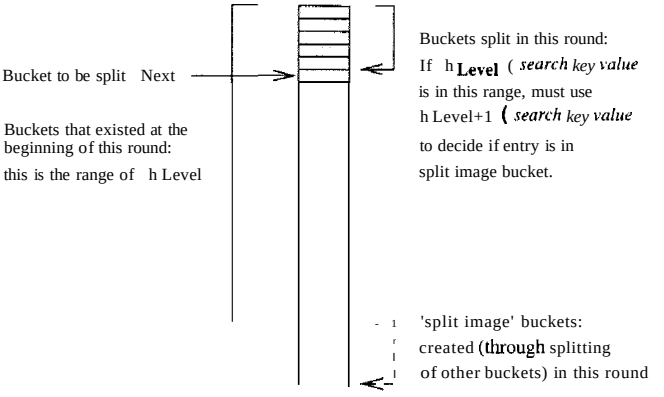


Figure 11.7 Buckets during a Round in Linear Hashing

Unlike Extendible Hashing, when an insert triggers a split, the bucket into which the data entry is inserted is not necessarily the bucket that is split. An overflow page is added to store the newly inserted data entry (which triggered the split), as in Static Hashing. However, since the bucket to split is chosen in round-robin fashion, eventually all buckets are split, thereby redistributing the data entries in overflow chains before the chains get to be more than one or two pages long.

We now describe Linear Hashing in more detail. A counter *Level* is used to indicate the current round number and is initialized to 0. The bucket to split is denoted by *Next* and is initially bucket 0 (the first bucket). We denote the number of buckets in the file at the beginning of round *Level* by N_{Level}. We can easily verify that $N_{Level} = N * 2^{Level}$. Let the number of buckets at the beginning of round 0, denoted by N_0, be N. We show a small linear hashed file in Figure 11.8. Each bucket can hold four data entries, and the file initially contains four buckets, as shown in the figure.

We have considerable flexibility in how to trigger a split, thanks to the use of overflow pages. We can split whenever a new overflow page is added, or we can impose additional conditions based all conditions such as space utilization. For our examples, a split is 'triggered' when inserting a new data entry causes the creation of an Qverftow page.

\Vhenever a split is triggered the *Next* bucket is split, and hash function $h_{Level+1}$ redistributes entries between this bucket (say bucket number b) and its split image; the split image is therefore bucket number $b + N_{Level}$. After splitting a bucket, the value of *Next* is incremented by 1. In the example file, insertion of

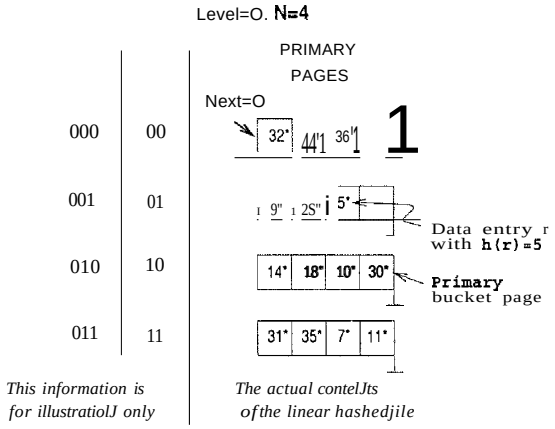


Figure 11.8 Example of a Linear Hashed File

data entry 43* triggers a split. The file after completing the insertion is shown in Figure 11.9.

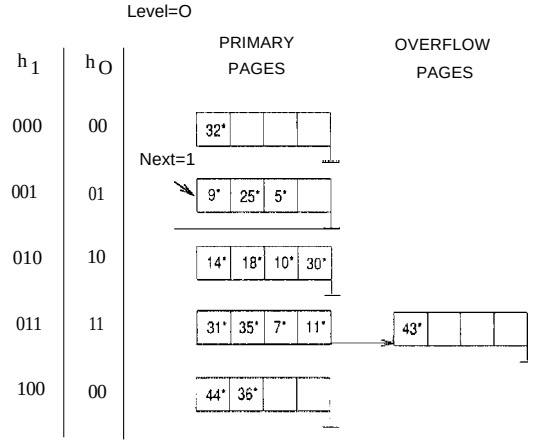


Figure 11.9 After Inserting Record r with $h(T) = 43$

At any time in .the middle of a round *Level,* all buckets above bucket *Next* have been split, and the file contains buckets that are their split images, as illustrated in Figure 11.7. Buckets *Next* through *NLevcl* have not yet been split. **If** we use *hLevel* on a data entry and obtain a number *b* in the range *Next* through *NLevel,* the data entry belongs to bucket *b.* For example, $h_O(18)$ is 2 (binary 10); since this value is between the current values of *Ne:r:t* (= 1) and N_1 (= 4), this bucket has not been split. However, if we obtain a number *b* in the range 0 through

Next, the data entry may be in this bucket or in its split image (which is bucket number $b + N_{Level}$); we have to use $h_{Level+1}$ to determine to which of these two buckets the data entry belongs. In other words, we have to look at one more bit of the data entry's hash value. For example, $h_0(32)$ and $h_0(44)$ are both a (binary 00). Since *Next* is currently equal to 1, which indicates a bucket that has been split, we have to apply hI' We have hI(32) = 0 (binary 000) and $h_1(44)$ = 4 (binary 100). Therefore, 32 belongs in bucket A and 44 belongs in its split image, bucket A2.

Not all insertions trigger a split, of course. If we insert 37* into the file shown in Figure 11.9, the appropriate bucket has space for the new data entry. The file after the insertion is shown in Figure 11.10.

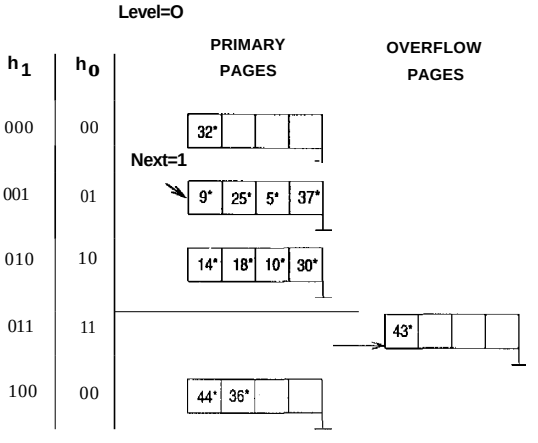


Figure 11.10 After Inserting Record r with $h(r)$ = 37

Sometimes the bucket pointed to by *Next* (the current candidate for splitting) is full, and a new data entry should be inserted in this bucket. In this case, a split is triggered, of course, but we do not need a new overflow bucket. This situation is illustrated by inserting 29* into the file shown in Figure 11.10. The result is shown in Figure 11.11.

When *Next* is equal to N_{Level} - 1 and a split is triggered, we split the last of the buckets present in the file at the beginning of round *Level.* The number of buckets after the split is twice the number at the beginning of the round, and we start a new round with *Level* incremented by 1 and *Next* reset to 0. Incrementing *Level* amounts to doubling the effective range into which keys are hashed. Consider the example file in Figure 11.12, which was obtained from the file of Figure 11.11 by inserting 22*, 66*, and 34*. (The reader is encouraged to try to work out the details of these insertions.) Inserting 50* causes a split that

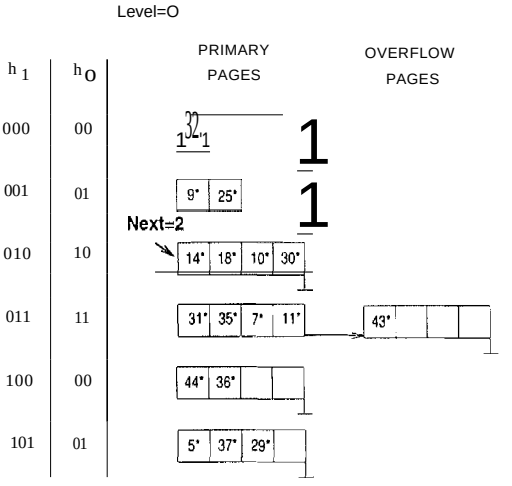


Figure 11.11 After Inserting Record r with $h(r^*) = 29$

leads to incrementing *Level,* as discussed previously; the file after this insertion is shown in Figure 11.13.

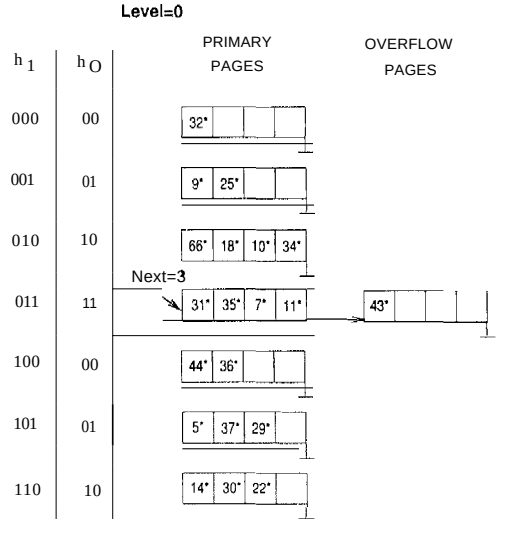


Figure 11.12 After Inserting Records with $h(r) = 22,66,and34$

In summary, an equality selection costs just one disk I/O unless the bucket has overflow pages; in practice, the cost on average is about 1.2 disk accesses for

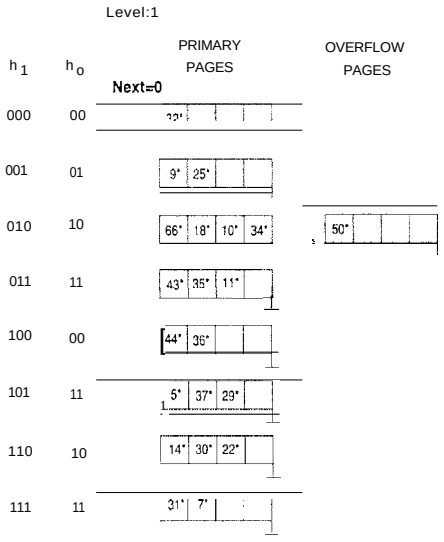


Figure 11.13 After Inserting Record r with $h(r) = 50$

reasonably uniform data distributions. (The cost can be considerably worse--linear in the number of data entries in the file----if the distribution is very skewed. The space utilization is also very poor with skewed data distributions.) Inserts require reading and writing a single page, unless a split is triggered.

We not discuss deletion in detail, but it is essentially the inverse of insertion. If the last bucket in the file is empty, it can be removed and *Next* can be decremented. (If *Next* is 0 and the last bucket becomes empty, *Next* is made to point to bucket $(M/2) - 1$, where *!vI* is the current number of buckets, *Level* is decremented, and the empty bucket is removed.) If we wish, we can combine the last bucket with its split image even when it is not empty, using some criterion to trigger this merging in essentially the same way. The criterion is typically based on the occupancy of the file, and merging can be done to improve space utilization.

11.4 EXTENDIBLE VS. LINEAR HASHING

To understand the relationship between Linear Hashing and Extendible Hashing, imagine that we also have a directory in Linear Hashing with elements 0 to $N - 1$. The first split is at bucket 0, and so we add directory element N. In principle, we may imagine that the entire directory has been doubled at this point; however, because element 1 is the same as element $N + 1$, elernent 2 is

the same as element $N + 2$, and so on, we can avoid the actual copying for the rest of the directory. The second split occurs at bucket 1; now directory element $N + 1$ becomes significant and is added. At the end of the round, all the original N buckets are split, and the directory is doubled in size (because all elements point to distinct buckets).

We observe that the choice of hashing functions is actually very similar to what goes on in Extendible Hashing---in effect, moving from h_i to h_{i+1} in Linear Hashing corresponds to doubling the directory in Extendible Hashing. Both operations double the effective range into which key values are hashed; but whereas the directory is doubled in a single step of Extendible Hashing, moving from h_i to h_{i+1}, along with a corresponding doubling in the number of buckets, occurs gradually over the course of a round in Linear Hashing. The new idea behind Linear Hashing is that a directory can be avoided by a clever choice of the bucket to split. On the other hand, by always splitting the appropriate bucket, Extendible Hashing may lead to a reduced number of splits and higher bucket occupancy.

The directory analogy is useful for understanding the ideas behind Extendible and Linear Hashing. However, the directory structure can be avoided for Linear Hashing (but not for Extendible Hashing) by allocating primary bucket pages consecutively, which would allow us to locate the page for bucket i by a simple offset calculation. For uniform distributions, this implementation of Linear Hashing has a lower average cost for equality selections (because the directory level is eliminated). For skewed distributions, this implementation could result in any empty or nearly empty buckets, each of which is allocated at least one page, leading to poor performance relative to Extendible Hashing, which is likely to have higher bucket occupancy.

A different implementation of Linear Hashing, in which a directory is actually maintained, offers the flexibility of not allocating one page per bucket; *null* directory elements can be used as in Extendible Hashing. However, this implementation introduces the overhead of a directory level and could prove costly for large, uniformly distributed files. (Also, although this implementation alleviates the potential problem of low bucket occupancy by not allocating pages for empty buckets, it is not a complete solution because we can still have many pages with very few entries.)

11.5 REVIEW **QUESTIONS**

Answers to the review questions can be found in the listed sections.

- How does a hash-based index handle an equality query? Discuss the use of the hash function in identifying a bucket to search. Given a bucket number, explain how the record is located on disk.
- Explain how insert and delete operations are handled in a static hash index. Discuss how overflow pages are used, and their impact on performance. How many disk I/Os does an equality search require, in the absence of overflow chains? What kinds of workload does a static hash index handle well, and when it is especially poor? **(Section 11.1)**
- How does Extendible Hashing use a directory of buckets? How does Extendible Hashing handle an equality query? How does it handle insert and delete operations? Discuss the *global depth* of the index and *local depth* of a bucket in your answer. Under what conditions can the directory can get large? **(Section 11.2)**
- What are *collisions?* Why do we need overflow pages to handle them? **(Section 11.2)**
- How does *Linear Hashing* avoid a directory? Discuss the round-robin splitting of buckets. Explain how the split bucket is chosen, and what triggers a split. Explain the role of the family of hash functions, and the role of the *Level* and *Next* counters. When does a round of splitting end? **(Section 11.3)**
- Discuss the relationship between Extendible and Linear Hashing. What are their relative merits? Consider space utilization for skewed distributions, the use of overflow pages to handle collisions in Extendible Hashing, and the use of a directory in Linear Hashing. **(Section 11.4)**

EXERCISES

Exercise **11.1** Consider the Extendible Hashing index shown in Figure 11.14. Answer the following questions about this index:

1. What can you say about the last entry that was inserted into the index?
2. What can you say about the last entry that was inserted into the index if you know that there have been no deletions from this index so far?
3. Suppose you are told that there have been no deletions from this index so far. What can you say about the last entry whose insertion into the index caused a split?
4. Show the index after inserting an entry with hash value 68.
5. Show the original index after inserting entries with hash values 17 and 69.
6. Show the original index after deleting the entry with hash value 21. (Assume that the full deletion algorithm is used.)
7. Show the original index after deleting the entry with hash value 10. Is a merge triggered by this deletion? If not, explain why. (Assume that the full deletion algorithm is used.)

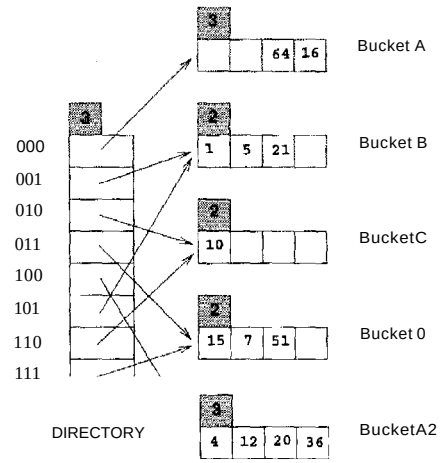


Figure 11.14 Figure for Exercise 11.1

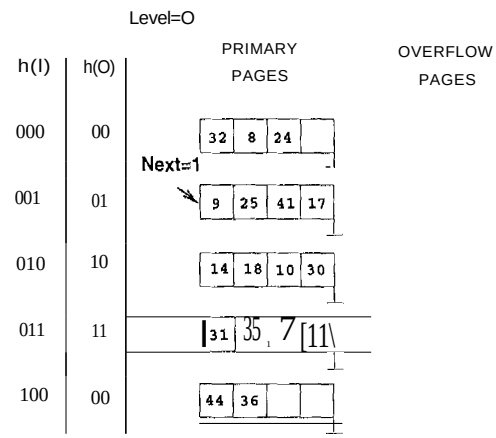


Figure 11.15 Figure for Exercise 11.2

Exercise 11.2 Consider the Linear Hashing index shown in Figure 11.15. Assume that we split whenever an overflow page is created. Answer the following questions about this index:

1. What can you say about the last entry that was inserted into the index?
2. What can you say about the last entry that was inserted into the index if you know that there llave been no deletions from this index so far?
3. Suppose you know that there have been no deletions from this index so far. What can you say about the last entry whose insertion into the index caused a split?
4. Show the index after inserting an entry with hash value 4.

5. Show the original index after inserting an entry with hash value 15.
6. Show the original index after deleting the entries with hash values 36 and 44. (Assume that the full deletion algorithm is used.)
7. Find a list of entries whose insertion into the original index would lead to a bucket with two overflow pages. Use as few entries as possible to accomplish this. What is the maximum number of entries that can be inserted into this bucket before a split occurs that reduces the length of this overflow chain?

Exercise 11.3 Answer the following questions about Extendible Hashing:

1. Explain why local depth and global depth are needed.
2. After an insertion that causes the directory size to double, how many buckets have exactly one directory entry pointing to them? If an entry is then deleted from one of these buckets, what happens to the directory size? Explain your answers briefly.
3. Does Extendible I-lashing guarantee at most one disk access to retrieve a record with a given key value?
4. If the hash function distributes data entries over the space of bucket numbers in a very skewed (non-uniform) way, what can you say about the size of the directory? What can you say about the space utilization in data pages (i.e., non-directory pages)?
5. Does doubling the directory require us to examine all buckets with local depth equal to global depth?
6. Why is handling duplicate key values in Extendible Hashing harder than in ISAM?

Exercise 11.4 Answer the following questions about Linear Hashing:

1. How does Linear Hashing provide an average-case search cost of only slightly more than one disk I/O, given that overflow buckets are part of its data structure?
2. Does Linear Hashing guarantee at most one disk access to retrieve a record with a given key value?
3. If a Linear Hashing index using Alternative (1) for data entries contains N records, with P records per page and an average storage utilization of 80 percent, what is the worst-case cost for an equality search? Under what conditions would this cost be the actual search cost?
4. If the hash function distributes data entries over the space of bucket numbers in a very skewed (non-uniform) way, what can you say about thc space utilization in data pages?

Exercise 11.5 Give an example of when you would use each element (A or B) for each of the following 'A versus B' pairs:

1. A hashed index using Alternative (1) versus heap file organization.
2. Extendible Hashing versus Linear Hashing.
3. Static Hashing versus Linear Hashing.
4. Static Hashing versus ISAIVI.
5. Linear Hashing versus B+ trees.

Exercise 11.6 Give examples of the following:

1. A Linear Hashing index and an Extendible Hashing index with the same data entries, such that the Linear Hashing index has more pages.

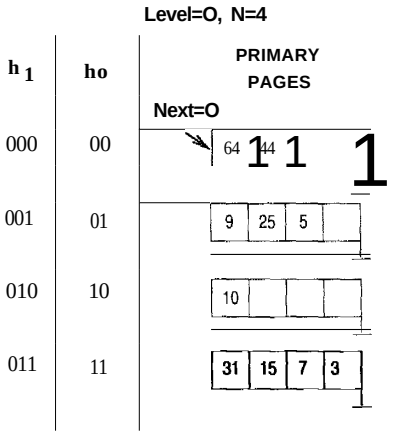


Figure **11.16** Figure for Exercise 11.9

2. A Linear Hashing index and an Extendible Hashing index with the same data entries, such that the Extendible Hashing index has more pages.

Exercise **11.7** Consider a relation R(*a*, *b*, c, *d*) containing 1 million records, where each page of the relation holds 10 records. R is organized as a heap file with unclustered indexes, and the records in R are randomly ordered. Assume that attribute *a* is a candidate key for R, with values lying in the range 0 to 999,999. For each of the following queries, name the approach that would most likely require the fewest I/Os for processing the query. The approaches to consider follow:

- Scanning through the whole heap file for R.
- Using a B+ tree index on attribute *R.a*.
- Using a hash index on attribute *R.a*.

The queries are:

1. Find all R tuples.
2. Find all R tuples such that $a < 50$.
3. Find all R tuples such that $a = 50$.
4. Find all R tuples such that $a > 50$ and $a < 100$.

Exercise 11.8 How would your answers to Exercise 11.7 change if *a* is not a candidate key for R? How would thcy change if we assume that records in R are sorted on *a*?

Exercise **11.9** Consider the snapshot of the Linear Hashing index shown in Figure 11.16. Assume that a bucket split occurs whcnever an overflow page is created.

1. What is the *maximum* number of data entries that call be inserted (given the best possible distribution of keys) before you have to split a bucket? Explain very briefly.
2. Show the file after inserting a *single* record whose insertion causes a bucket split.

3. (a) What is the *minimum* number of record insertions that will cause a split of all four buckets? Explain very briefly.
   (b) What is the value of *Next* after making these insertions?
   (c) What can you say about the number of pages in the fourth bucket shown after this series of record insertions?

Exercise **11.10** Consider the data entries in the Linear Hashing index for Exercise 11.9.

1. Show an Extendible Hashing index with the same data entries.
2. Answer the questions in Exercise 11.9 with respect to this index.

Exercise **11.11** In answering the following questions, assume that the full deletion algorithm is used. Assume that merging is done when a bucket becomes empty.

1. Give an example of Extendible Hashing where deleting an entry reduces global depth.
2. Give an example of Linear Hashing in which deleting an entry decrements *Next* but leaves *Level* unchanged. Show the file before and after the deletion.
3. Give an example of Linear Hashing in which deleting an entry decrements *Level*. Show the file before and after the deletion.
4. Give an example of Extendible Hashing and a list of entries e1, *e2, e3* such that inserting the entries in order leads to three splits and deleting them in the reverse order yields the original index. If such an example does not exist, explain.
5. Give an example of a Linear Hashing index and a list of entries e1, *e2, e3* such that inserting the entries in order leads to three splits and deleting them in the reverse order yields the original index. If such an example does not exist, explain.

PROJECT-BASED EXERCISES

Exercise **11.12** *(Note to instructors: Additional details must be provided if this question is assigned. See Appendix 30.)* Implement Linear Hashing or Extendible Hashing in Minibase.

BIBLIOGRAPHIC NOTES

Hashing is discussed in detail in [442]. Extendible Hashing is proposed in [256]. Litwin proposed Linear Hashing in [483]. A generalization of Linear Hashing for distributed environments is described in [487]. There has been extensive research into hash-based indexing techniques. Larson describes two variations of Linear Hashing in [469] and [470]. Ramakrishna presents an analysis of hashing techniques in [607]. Hash functions that do not produce bucket overflows are studied in [608]. Order-preserving hashing techniques are discussed in [484] and [308]. Partitioned-hashing, in which each field is hashed to obtain some bits of the bucket address, extends hashing for the case of queries in which equality conditions are specified only for some of the key fields. This approach was proposed by Rivest [628] and is discussed in [747]; a further development is described in [616].

PARTN

QUERY EVALUATION

12

OVERVIEW OF QUERY EVALUATION

- What descriptive information does a DBMS store in its catalog?
- What alternatives are considered for retrieving rows from a table?
- Why does a DBMS implement several algorithms for each algebra operation? What factors affect the relative performance of different algorithms?
- What are query evaluation plans and how are they represented?
- Why is it important to find a good evaluation plan for a query? How is this done in a relational DBMS?
- Key concepts: catalog, system statistics; fundamental techniques, indexing, iteration, and partitioning; access paths, matching indexes and selection conditions; selection operator, indexes versus scans, impact of clustering; projection operator, duplicate elimination; join operator, index nested-loops join, sort-merge join; query evaluation plan; materialization vs. pipelinining; iterator interface; query optimization, algebra equivalences, plan enumeration; cost estimation

This very remarkable man, commends a most practical plan:
You can do what you want, if you don't think you can't,
So c1on't think you can't if you can.

—Charles Inge

In this chapter, we present an overview of how queries are evaluated in a relational DBMS. We begin with a discussion of how a DBMS describes the data

that it manages, including tables and indexes, in Section 12.1. This descriptive data, or metadata, stored in special tables called the system catalogs, is used to find the best way to evaluate a query.

SQL queries are translated into an extended form of relational algebra, and query evaluation plans are represented as trees of relational operators, along with labels that identify the algorithm to use at each node. Thus, relational operators serve as building blocks for evaluating queries, and the implementation of these operators is carefully optimized for good performance. We introduce operator evaluation in Section 12.2 and describe evaluation algorithms for various operators in Section 12.3.

In general, queries are composed of several operators, and the algorithms for individual operators can be combined in many ways to evaluate a query. The process of finding a good evaluation plan is called *query optimization*. We introduce query optimization in Section 12.4. The basic task in query optimization, which is to consider several alternative evaluation plans for a query, is motivated through examples in Section 12.5. In Section 12.6, we describe the space of plans considered by a typical relational optimizer.

The ideas are presented in sufficient detail to allow readers to understand how current database systems evaluate typical queries. This chapter provides the necessary background in query evaluation for the discussion of physical database design and tuning in Chapter 20. Relational operator implementation and query optimization are discussed further in Chapters 13, 14, and 15; this in-depth coverage describes how current systems are implemented.

We consider a number of example queries using the following schema:

Sailors(sid: integer, *sname:* string, *rating:* integer, *age:* real)
Reserves(sid: integer, *bid:* integer, *day:* dates, *marne:* string)

We aSSUlne that each tuple of Reserves is 40 bytes long, that a page can hold 100 Reserves tuples, and that we have 1000 pages of such tuples. Similarly, we assume that each tuple of Sailors is 50 bytes long, that a page can hold 80 Sailors tuples, and that we have 500 pages of such tuples.

12.1 THE SYSTEM CATALOG

We can store a table using one of several alternative file structures, and we can create one or more indexes -each stored as a file 011 every tal)le. Conversely, in a relational DBMS, every file contains either the tuples in a table or the

entries in an index. The collection of files corresponding to users' tables and indexes represents the *data* in the database.

A relational DBMS maintains information about every table and index that it contains. The descriptive information is itself stored in a collection of special tables called the catalog tables. An example of a catalog table is shown in Figure 12.1. The catalog tables are also called the data dictionary, the system catalog, or simply the *catalog.*

12.1.1 Information in the Catalog

Let us consider what is stored in the system catalog. At a minimum, we "have system-wide information, such as the size of the buffer pool and the page size, and the following information about individual tables, indexes, and views:

- For each table:
  - Its *table name,* the *file name* (or some identifier), and the *file structure* (e.g., heap file) of the file in which it is stored.
  - The *attribute name* and *type* of each of its attributes.
  - The *index name* of each index on the table.
  - The *integrity constmints* (e.g., primary key and foreign key constraints) on the table.
- For each index:
  - The *index name* and the *structure* (e.g., B+ tree) of the index.
  - The *search key* attributes.
- For each view:
  - Its *view name* and *definition.*

In addition, statistics about tables and indexes are stored in the system catalogs and updated periodically *(not* every time the underlying tables are modified). The following information is commonly stored:

- Cardinality: The number of tuples *NTuples(R)* for each table *R.*
- Size: The number of pages *NPages(R)* for each table *R.*
- Index Cardinality: The number of distinct key values *NKeys(I)* for each index *I.*
- Index Size: The nUluber of pages *INPages(I)* for each index *I.* (For a B+ tree index *I,* we take *INPagcs* to be the number of leaf pages.)

- **Index Height:** The number of nonleaf levels *IHe'ight(I)* for each tree index *I*.
- **Index Range:** The minimum present key value *ILow(I)* and the maximum present key value *IHigh(I)* for each index *I*.

We assume that the database architecture presented in Chapter 1 is used. Further, we assume that each file of records is implemented as a separate file of pages. Other file organizations are possible, of course. For example, a page file can contain pages that store records from more than one record file. If such a file organization is used, additional statistics must be maintained, such as the fraction of pages in a file that contain records from a given collection of records.

The catalogs also contain information about *users*, such as accounting information and *authorization* information (e.g., Joe User can modify the Reserves table but only read the Sailors table).

How Catalogs are Stored

An elegant aspect of a relational DBMS is that the system catalog is itself a collection of tables. For example, we might store information about the attributes of tables in a catalog table called Attribute_Cat:

Attribute_Cat(*attr_name:* string, *rel_name:* string,
type: string, *position:* integer)

Suppose that the database contains the two tables that we introduced at the begining of this chapter:

Sailors(sid: integer, *sname:* string, *rating:* integer, *age:* real)
Reserves(sid: integer, *bid:* integer, *day:* dates, *mame:* string)

Figure 12.1 shows the tuples in the Attribute_Cat table that describe the attributes of these two tables. Note that in addition to the tuples describing Sailors and Reserves, other tuples (the first four listed) describe the four attributes of the Attribute_Cat table itself! These other tuples illustrate an important Point: the catalog tables describe all the tables in the database, *including* the catalog tables themselves. When information about a table is needed, it is obtained from the system catalog. Of course, at the implementation level, whenever the DBMS needs to find the schema of a *catalog* table, the code that retrieves this information must be handled specially. (Otherwise, the code has to retrieve this information from the catalog tables without, presumably, knowing the schema of the catalog tables.)

attr_name	*rel*	*type*	
attr_name	Attribute_Cat	string	1
reLname	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Sailors	integer	1
sname	Sailors	string	2
rating	Sailors	integer	3
age	Sailors	real	4
sid	Reserves	integer	1
bid	Reserves	integer	2
day	Reserves	dates	3
rname	Reserves	string	4

Figure 12.1 An Instance of the Attribute_Cat Relation

The fact that the system catalog is also a collection of tables is very useful. For example, catalog tables can be queried just like any other table, using the query language of the DBMS! Further, all the techniques available for implementing and managing tables apply directly to catalog tables. The choice of catalog tables and their schemas is not unique and is made by the implementor of the DBMS. Real systems vary in their catalog schema design, but the catalog is always implemented as a collection of tables, and it essentially describes all the data stored in the database.[1]

12.2 INTRODUCTION TO OPERATOR EVALUATION

Several alternative algorithms are available for implementing each relational operator, and for most operators no algorithm is universally superior. Several factors influence which algorithm performs best, including the sizes of the tables involved, existing indexes and sort orders, the size of the available buffer pool, and the buffer replacement policy.

In this section, we describe some common techniques used in developing evaluation algorithms for relational operators, and introduce the concept of *access paths*, which are the different ways in which rows of a table can be retrieved.

[1]Some systems may store additional information in a non-relational form. For example, a system with a sophisticated query optimizer may maintain histograms or other statistical information about the distribution of values in certain attributes of a table. We can think of such information, when it is maintained, as a supplement to the catalog tables.

12.2.1 Three Common Techniques

The algorithms for various relational operators actually have a lot in common. A few simple techniques are used to develop algorithms for each operator:

- Indexing: If a selection or join condition is specified, use an index to examine just the tuples that satisfy the condition.
- Iteration: Examine all tuples in an input table, one after the other. If we need only a few fields from each tuple and there is an index whose key contains all these fields, instead of examining data tuples, we can scan all index data entries. (Scanning all data entries sequentially makes no use of the index's hash- or tree-based search structure; in a tree index, for example, we would simply examine all leaf pages in sequence.)
- Partitioning: By partitioning tuples on a sort key, we can often decompose an operation into a less expensive collection of operations on partitions. *Sorting* and *hashing* are two commonly used partitioning techniques.

We discuss the role of indexing in Section 12.2.2. The iteration and partitioning techniques are seen in Section 12.3.

12.2.2 Access Paths

An access path is a way of retrieving tuples from a table and consists of either (1) a file scan or (2) an index plus a *matching* selection condition. Every relational operator accepts one or more tables as input, and the access methods used to retrieve tuples contribute significantly to the cost of the operator.

Consider a simple selection that is a conjunction of conditions of the form $attr$ op $value$, where op is one of the comparison operators $<$, $\leq$, $=$, $\neq$, $\geq$, or $>$. Such selections are said to be in conjunctive normal form (CNF), and each condition is called a conjunct.[2] Intuitively, an index matches a selection condition if the index can be used to retrieve just the tuples that satisfy the condition.

- A hash index matches a CNF selection if there is a term of the form $attribute$=$value$ in the selection for each attribute in the index's search key.
- A tree index matches a CNF selection if there is a term of the form $attribute$ op $value$ for each attribute in a *prefix* of the index's search key. ($\langle a \rangle$ and $\langle a, b \rangle$ are prefixes of key (a,b,e), but $\langle a, c \rangle$ and $\langle b, c \rangle$ are not.)

[2] We consider more complex selection conditions in Section 14.2.

Note that **op** can be any comparison; it is not restricted to he equality as it is for matching selections on a hash index.

An index can match some subset of the conjuncts in a selection condition (in CNP), even though it does not match the entire condition. We refer to the conjuncts that the index matches as the **primary conjuncts** in the selection.

The following examples illustrate access paths.

- If we have a hash index H on the search key $\langle rname,bid,sid\rangle$, we can use the index to retrieve just the Sailors tuples that satisfy the condition *rnarne='Joe'l\ bid=5 1\ sid=3.* The index matches the entire condition *rname='Joe' 1\ bid=5* $\wedge$ *sid=* 3. On the other hand, if the selection condition is *rname='Joe' 1\ bid=5,* or some condition on *date,* this index does not match. That is, it cannot be used to retrieve just the tuples that satisfy these conditions.

  In contrast, if the index were a B+ tree, it would match both *rname='Joe' 1\ bid=51\ 8id=3* and *mame='Joe' 1\ bid=5.* However, it would not match *bid=5 1\ sid=3* (since tuples are sorted primarily by *rnarne).*

- If we have an index (hash or tree) on the search key $\langle bid,sid\rangle$ and the selection condition *rname='Joe' 1\ bid=5 1\ sid=3,* we can use the index to retrieve tuples that satisfy *bid=51\ sid=3;* these are the primary conjuncts. The fraction of tuples that satisfy these conjuncts (and whether the index is clustered) determines the number of pages that are retrieved. The additional condition on *Tna7ne* must then be applied to each retrieved tuple and will eliminate some of the retrieved tuples from the result.

- If we have an index on the search key $\langle bid, sid\rangle$ and we also have a B+ tree index on *day,* the selection condition *day* < *8/9/2002 1\ bid=5* $\wedge$ *sid=3* offers us a choice. Both indexes match (part of) the selection condition, and we can use either to retrieve Reserves tuples. \Vhichever index we use, the conjuncts in the selection condition that are not matched by the index (e.g., *bid=51\ sid=3* if we use the B+ tree index on *day*) must be checked for each retrieved tuple.

Selectivity of Access Paths

The selectivity of an access path is the number of pages retrieved (index pages plus data pages) if we usc this access path to retrieve all desired tuples. If a table contains an index that matches a given selection, there are at least two access paths: the index and a scan of the data file. Sometimes, of course, we can scan the index itself (rather than scanning the data file or using the index to probe the file), giving us a third access path.

The most selective access path is the one that retrieves the fewest pages; using the most selective access path minimizes the cost of data retrieval. The selectivity of an access path depends on the primary conjuncts in the selection condition (with respect to the index involved). Each conjunct acts as a filter on the table. The fraction of tuples in the table that satisfy a given conjunct is called the reduction factor. 'When there are several primary conjuncts, the fraction of tuples that satisfy all of them can be approximated by the product of their reduction factors; this effectively treats them as independent filters, and while they may not actually be independent, the approximation is widely used in practice.

Supose we have a hash index H on Sailors with search key $\langle rname, bid, sid \rangle$, and we are given the selection condition *rname='Joe' 1\ bid=5 1\ sid=3.* We can use the index to retrieve tuples that satisfy all three conjuncts. The catalog contains the number of distinct key values, $NKeys(H)$, in the hash index, as well as the number of pages, $NPages$, in the Sailors table. The fraction of pages satisfying the primary conjuncts is $Npages(Sailors) \cdot \frac{1}{NKeys(H)}$.

If the index has search key $\langle bid, sid \rangle$, the primary conjuncts are *bid=51\ sid=3.* If we know the number of distinct values in the *bid* column, we can estimate the reduction factor for the first conjunct. This information is available in the catalog if there is an index with *bid* as the search key; if not, optimizers typically use a default value such as 1/10. Multiplying the reduction factors for *bid=5* and *sid=3* gives us (under the simplifying independence assumption) the fraction of tuples retrieved; if the index is clustered, this is also the fraction of pages retrieved. If the index is not clustered, each retrieved tuple could be on a different page. (Review Section 8.4 at this time.)

We estimate the reduction factor for a range condition such as *day> 8/9/2002* by assuming that values in the column are uniformly distributed. If there is a B+ tree T with key day, the reduction factor is $\frac{High(T) - value}{High(T) - Low(T)}$.

12.3 ALGORITHMS FOR RELATIONAL OPERATIONS

We now briefly discuss evaluation algorithms for the main relational operators. While the important ideas are introduced here, a more in-depth treatment is deferred to Chapter 14. As in Chapter 8, we consider only I/O costs and measure I/O costs in terms of the number of page I/Os. In this chapter, we use detailed examples to illustrate how to compute the cost of an algorithm. Although we do not present rigorous cost formulas in this chapter, the reader should be able to apply the underlying icleas to do cost calculations on other similar examples.

12.3.1 Selection

The selection operation is a simple retrieval of tuples from a table, and its implementation is essentially covered in our discussion of access paths. To summarize, given a selection of the form $\sigma_{R.attr\ \mathbf{op}\ value}(R)$, if there is no index on *R.attr,* we have to scan *R.*

If one or more indexes on *R* match the selection, we can use the index to retrieve matching tuples, and apply any remaining selection conditions to further restrict the result set. As an example, consider a selection of the form *rname* < *'C%'* on the Reserves table. Assuming that names are uniformly distributed with respect to the initial letter, for simplicity, we estimate that roughly 10% of Reserves tuples are in the result. This is a total of 10,000 tuples, or 100 pages. If we have a clustered B+ tree index on the *rname* field of Reserves, we can retrieve the qualifying tuples with 100 l/Os (plus a few l/Os to traverse from the root to the appropriate leaf page to start the scan). However, if the index is unclustered, we could have up to 10,000 l/Os in the worst case, since each tuple could cause us to read a page.

As a rule of thumb, it is probably cheaper to simply scan the entire table (instead of using an unclustered index) if over 5% of the tuples are to be retrieved.

Sec Section 14.1 for more details on implementation of selections.

12.3.2 Projection

The projection operation requires us to drop certain fields of the input, which is easy to do. The expensive aspect of the operation is to ensure that no duplicates appear in the result. For example, if we only want the *sid* and *bid* fields from Reserves, we could have duplicates if a sailor has reserved a given boat on several days.

If duplicates need not be eliminated (e.g., the DISTINCT keyword is not included in the SELECT clause), projection consists of simply retrieving a subset of fields from each tuple of the input table. This can be accomplished by simple iteration on either the table or an index whose key contains all necessary fields. (Note that we do not care whether the index is clustered, since the values we want are in the data entries of the index itself!)

If we have to eliminate duplicates, we typically have to use partitioning. Suppose we want to obtain $\langle sid, bid \rangle$ by projecting from Reserves. We can partition by (1) scanning H.eserves to obtain *(sid, bid)* pairs and (2) sorting these pairs

using ⟨*sid, bid*⟩ as the sort key. We can then scan the sorted pairs and easily discard duplicates, which are now adjacent.

Sorting large disk-resident datasets is a very important operation in database systems, and is discussed in Chapter 13. Sorting a table typically requires two or three passes, each of which reads and writes the entire table.

The projection operation can be optimized by combining the initial scan of Reserves with the scan in the first pass of sorting. Similarly, the scanning of sorted pairs can be combined with the last pass of sorting. With such an optimized implemention, projection with duplicate elimination requires (1) a first pass in which the entire table is scanned, and only pairs ⟨*sid, bid*⟩ are written out, and (2) a final pass in which all pairs are scanned, but only one copy of each pair is written out. In addition, there might be an intermediate pass in which all pairs are read from and written to disk.

The availability of appropriate indexes can lead to less expensive plans than sorting for duplicate elimination. If we have an index whose search key contains all the fields retained by the projection, we can sort the data entries in the index, rather than the data records themselves. If all the retained attributes appear in a prefix of the search key for a clustered index, we can do even better; we can simply retrieve data entries using the index, and duplicates are easily detected since they are adjacent. These plans are further examples of *index-only* evaluation strategies, which we discussed in Section 8.5.2.

See Section 14.3 for more details on implementation of projections.

12.3.3 Join

Joins are expensive operations and very common. Therefore, they have been widely studied, and systems typically support several algorithms to carry out joins.

Consider the join of Reserves and Sailors, with the join conclition *Reserves.sid* = *Sailors.sid*. Suppose that one of the tables, say Sailors, has an index on the *sid* column. We can scan Reserves and, for each tuple, use the index to *pTObe* Sailors for matching tuples. This approach is called index nested loops join.

Suppose that we have a hash-based index using Alternative (2) on the *sid* attribute of Sailors and that it takes about 1.2 1/0s on average[3] to retrieve the appropriate page of the index. Since *sid* is a key for Sailors, we have at

:IThis is a typical cost for hash-based indexes.

most one matching tuple, Indeed, *sid* in Reserves is a foreign key referring to Sailors, and therefore we have *exactly* one matching Sailors tuple for each Reserves tuple, Let us consider the cost of scanning Reserves and using the index to retrieve the matching Sailors tuple for each Reserves tuple, The cost of scanning Reserves is 1000. There are 100 * 1000 tuples in Reserves. For each of these tuples, retrieving the index page containing the rid of the matching Sailors tuple costs 1.2 I/Os (on average); in addition, we have to retrieve the Sailors page containing the qualifying tuple, Therefore, we have 100,000 * (1 + 1.2) I/Os to retrieve matching Sailors tuples. The total cost is 221,000 I/Os.[4]

If we do not have an index that matches the join condition on either table, we cannot use index nested loops, In this case, we can sort both tables on the join column, and then scan them to find matches. This is called sort-merge join.. Assuming that we can sort Reserves in two passes, and Sailors in two passes as well, let us consider the cost of sort-merge join. Consider the join of the tables Reserves and Sailors. Because we read and write Reserves in each pass, the sorting cost is 2 * 2 * 1000 = 4000 I/Os. Similarly, we can sort Sailors at a cost of 2 * 2 * 500 = 2000 I/Os. In addition, the second phase of the sort-merge join algorithm requires an additional scan of both tables. Thus the total cost is 4000 + 2000 + 1000 + 500 = 7500 I/Os.

Observe that the cost of sort-merge join, which does not require a pre-existing index, is lower than the cost of index nested loops join, In addition, the result of the sort-merge join is sorted on the join column(s). Other join algorithms that do not rely on an existing index and are often cheaper than index nested loops join are also known *(block nested loops* and *hash* joins; see Chapter 14). Given this, why consider index nested loops at all?

Index nested loops has the nice property that it is incremental. The cost of our example join is incremental in the number of Reserves tuples that we process. Therefore, if some additional selection in the query allows us to consider only a small subset of Reserves tuples, we can avoid computing the join of Reserves and Sailors in its entirety. For instance, suppose that we only want the result of the join for boat 101, and there are very few such reservations. For each such Reserves tuple, we probe Sailors, and we are clone. If we use sort-merge join, on the other hand, we have to scan the entire Sailors table at least once, and the cost of this step alone is likely to be much higher than the entire cost of index nested loops join.

Observe that the choice of index nested loops join is based on considering the query as a whole, including the extra selection all Reserves, rather than just

[4]As an exercise, the reader should write formulas for the cost estimates in this example in terms of the properties e.g., NPages-of the tables and indexes involved.

the join operation by itself. This leads us to our next topic, query optimization, which is the process of finding a good plan for an entire query.

See Section 14.4 for more details.

12.3.4 Other Operations

A SQL query contains group-by and aggregation in addition to the basic relational operations. Different query blocks can be combined with union, set-difference, and set-intersection.

The expensive aspect of set operations such as union and intersection is duplicate elimination, just like for projection. The approach used to implement projection is easily adapted for these operations as well. See Section 14.5 for more details.

Group-by is typically implemented through sorting. Sometimes, the input table has a tree index with a search key that matches the grouping attributes. In this case, we can retrieve tuples using the index in the appropriate order without an explicit sorting step. Aggregate operations are carried out using temporary counters in main memory as tuples are retrieved. See Section 14.6 for more details.

12.4 INTRODUCTION TO QUERY OPTIMIZATION

Query optimization is one of the most important tasks of a relational DBMS. One of the strengths of relational query languages is the wide variety of ways in which a user can express and thus the system can evaluate a query. Although this flexibility makes it easy to write queries, good performance relies greatly on the quality of the query optimizer—a given query can be evaluated in many ways, and the difference in cost between the best and worst plans may be several orders of magnitude. Realistically, we cannot exped to always find the best plan, but we expect to consistently find a plan that is quite good.

A more detailed view of the query optimization and execution layer in the DBMS architecture from Section 1.8 is shown in Figure 12.2. Queries are parsed and then presented to a **query optimizer,** which is responsible for identifying an efficient execution plan. The optimizer generates alternative plans and chooses the plan wit.h the least estimated cost.

The space of plans considered by a typical relational query optimizer can be understood by recognizing that *a query is essentially treated as a* σ - π - $\bowtie$ *algebra expression,* with the remaining operations (if any, in a given query)

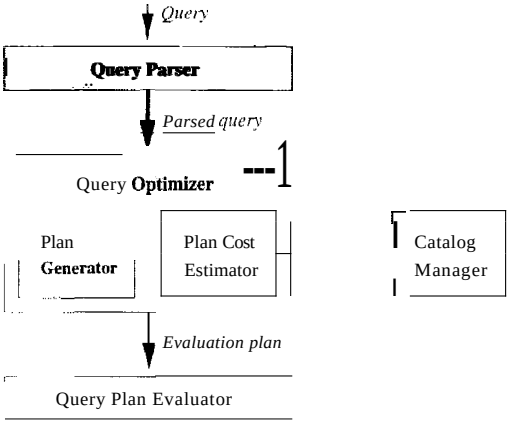


Figure 12.2 Query Parsing, Optimization, and Execution

CommercialOptimizers: Current relational DBMS optimizers are very complex pieces of software with many closely guarded details, and they typically represent 40 to 50 man-years of development effort!

carried out on the result of the σ - π— [x] expression. Optimizing such a relational algebra expression involves two basic steps:

- Enumerating alternative plans for evaluating the expression. Typically, an optimizer considers a subset of all possible plans because the number of possible plans is very large.
- Estimating the cost of each enumerated plan and choosing the plan with the lowest estimated cost.

In this section we lay the foundation for our discussion of query optimization by introducing evaluation plans.

12.4.1 Query Evaluation Plans

A query evaluation plan (or simply plan) consists of an extended relational algebra tree, with additional annotations at each node indicating the access methods to use for each table and the implementation method to use for each relational operator.

Consider the following SQL query:

```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
       AND R.bid = 100 AND S.rating > 5
```

This query can be expressed in relational algebra as follows:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

This expression is shown in the form of a tree in Figure 12.3. The algebra expression partially specifies how to evaluate the query-owe first compute the natural join of Reserves and Sailors, then perform the selections, and finally project the *snarne* field.

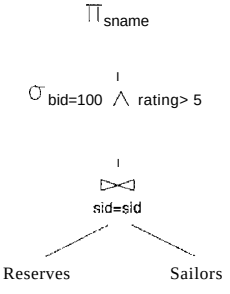


Figure 12.3 Query Expressed as a Relational Algebra Tree

To obtain a fully specified evaluation plan, we must decide on an implementation for each of the algebra operations involved. For example, we can use a page-oriented simple nested loops join with Reserves as the outer table and apply selections and projections to each tuple in the result of the join as it is produced; the result of the join before the selections and projections is never stored in its entirety. This query evaluation plan is shown in Figure 12.4.

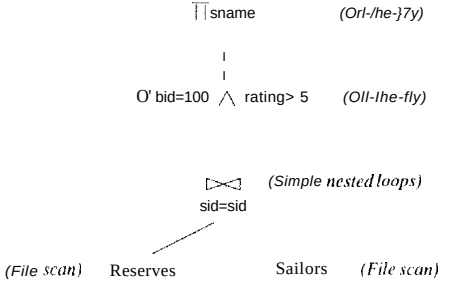


Figure 12.4 Query Evaluation Plan for Sample Query

In drawing the query evaluation plan, we have used the convention that the *outer table* is the *left child* of the join operator. We adopt this convention henceforth.

12.4.2 Multi-operator Queries: Pipelined Evaluation

When a query is composed of several operators, the result of one operator is sometimes pipelined to another operator without creating a temporary table to hold the intermediate result. The plan in Figure 12.4 pipelines the output of the join of Sailors and Reserves into the selections and projections that follow. Pipelining the output of an operator into the next operator saves the cost of writing out the intermediate result and reading it back in, and the cost savings can be significant. If the output of an operator is saved in a temporary table for processing by the next operator, we say that the tuples are materialized. Pipelined evaluation has lower overhead costs than materialization and is chosen whenever the algorithm for the operator evaluation permits it.

There are many opportunities for pipelining in typical query plans, even simple plans that involve only selections.. Consider a selection query in which only part of the selection condition matches an index. We can think of such a query as containing *two* instances of the selection operator: The first contains the primary, or matching, part of the original selection condition, and the second contains the rest of the selection condition. We can evaluate such a query by applying the primary selection and writing the result to a temporary table and then applying the second selection to the temporary table. In contrast, a pipelined evaluation consists of applying the second selection to each tuple in the result of the primary selection as it is produced and adding tuples that qualify to the final result. When the input table to a unary operator (e.g., selection or projection) is pipelined into it, we sometimes say that the operator is applied on-the-fly.

As a second and more general example, consider a join of the form $(A \bowtie B) \bowtie C$, shown in Figure 12.5 as a tree of join operations.

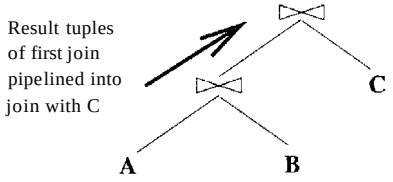


Figure 12.5 A Query Tree Illustrating Pipelilling

Both joins can be evaluated in pipelined fashion using some version of a nested loops join. Conceptually, the evaluation is initiated from the root, and the node joining A and B produces tuples as and when they are requested by its parent node. 'When the root node gets a page of tuples from its left child (the outer table), all the matching inner tuples are retrieved (using either an index or a scan) and joined with matching outer tuples; the current page of outer tuples is then discarded. A request is then made to the left child for the next page of tuples, and the process is repeated. Pipelined evaluation is thus a *control strategy* governing the rate at which different joins in the plan proceed. It has the great virtue of not writing the result of intermediate joins to a temporary file because the results are produced, consumed, and discarded one page at a time.

12.4.3 The Iterator Interface

A query evaluation plan is a tree of relational operators and is executed by calling the operators in some (possibly interleaved) order. Each operator has one or more inputs and an output, which are also nodes in the plan, and tuples must be passed between operators according to the plan's tree structure.

To simplify the code responsible for coordinating the execution of a plan, the relational operators that form the nodes of a plan tree (which is to be evaluated using pipelining) typically support a uniform **iterator** interface, hiding the internal implementation details of each operator. The iterator interface for an operator includes the functions open, geLnext, and close. The *open* function initializes the state of the iterator by allocating buffers for its inputs and output, and is also used to pass in arguments such as selection conditions that modify the behavior of the operator. The code for the *get_next* function calls the *get_next* function on each input node and calls operator-specific code to process the input tuples. The output tuples generated by the processing are placed in the output buffer of the operator, and the state of the iterator is updated to keep track of how much input has been consumed. When all output tuples have been produced through repeated calls to *get_next*, the *close* function is called (by the code that initiated execution of this operator) to deallocate state information.

The iterator interface supports pipelining of results naturally: the decision to pipeline or materialize input tuples is encapsulated in the operator-specific code that processes input tuples. If the algorithm implemented for the operator allows input tuples to be processed completely when they are received, input tuples are not Inaterialized and the evaluation is pipelined. If the algorithm examines the same input tuples several times, they are materialized. This

decision, like other details of the operator's implementation, is hidden by the iterator interface for the operator.

The iterator interface is also used to encapsulate access methods such as B+ trees and hash-based indexes. Externally, access methods can be viewed simply as operators that produce a stream of output tuples. In this case, the *open* function can be used to pass the selection conditions that match the access path.

12.5 ALTERNATIVE PLANS: A MOTIVATING EXAMPLE

Consider the example query from Section 12.4. Let us consider the cost of evaluating the plan shown in Figure 12.4. We ignore the cost of writing out the final result since this is common to all algorithms, and does not affect their relative costs. The cost of the join is 1000 + 1000 * 500 = 501,000 page I/Os. The selections and the projection are done on-the-fly and do not incur additional I/Os. The total cost of this plan is therefore 501,000 page I/Os. This plan is admittedly naive; however, it is possible to be even more naive by treating the join as a cross-product followed by a selection.

We now consider several alternative plans for evaluating this query. Each alternative improves on the original plan in a different way and introduces some optimization ideas that are examined in more detail in the rest of this chapter.

12.5.1 Pushing Selections

A join is a relatively expensive operation, and a good heuristic is to reduce the sizes of the tables to be joined as much as possible. One approach is to apply selections early; if a selection operator appears after a join operator, it is worth examining whether the selection can be 'pushed' ahead of the join. As an example, the selection *bid=100* involves only the attributes of Reserves and can be applied to Reserves *before* the join. Similarly, the selection *rating> 5* involves only attributes of Sailors and can be applied to Sailors before the join. Let us suppose that the selections are performed using a simple file scan, that the result of each selection is written to a temporary table on disk, and that the temporary tables are then joined using a sort-merge join. The resulting query evaluation plan is shown in Figure 12.6.

Let us assume that five buffer pages are available and estimate the cost of this query evaluation plan. (It is likely that more buffer pages are available in practice. We chose a small number simply for illustration in this example.) The cost of applying *bid=100* to Reserves is the cost of scanning Reserves (1000 pages) plus the cost of writing the result to a temporary table, say T1.

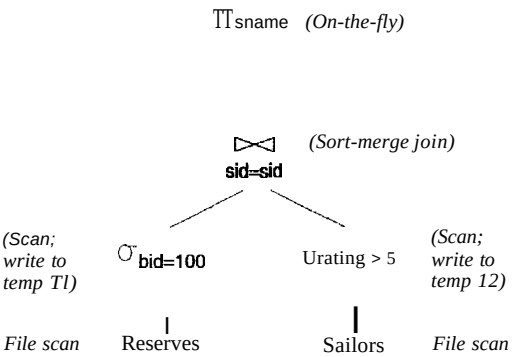


Figure 12.6 A Second Query Evaluation Plan

(Note that the cost of writing the temporary table cannot be ignored-we can ignore only the cost of writing out the *final* result of the query, which is the only component of the cost that is the same for all plans.) To estimate the size of T1, we require additional information. For example, if we assume that the maximum number of reservations of a given boat is one, just one tuple appears in the result. Alternatively, if we know that there are 100 boats, we can assume that reservations are spread out uniformly across all boats and estimate the number of pages in T1 to be 10. For concreteness, assume that the number of pages in T1 is indeed 10.

The cost of applying *rating* > 5 to Sailors is the cost of scanning Sailors (500 pages) plus the cost of writing out the result to a temporary table, say, T2. If we assume that ratings are uniformly distributed over the range 1 to 10, we can approximately estimate the size of T2 as 250 pages.

To do a sort-merge join of T1 and T2, let us assume that a straightforward implementation is used in which the two tables are first completely sorted and then merged. Since five buffer pages are available, we can sort T1 (which has 10 pages) in two passes. Two runs of five pages each are produced in the first pass and these are merged in the second pass. In each pass, we read and write 10 pages; thus, the cost of sorting T1 is $2 * 2 * 10 = 40$ page I/Os. We need four passes to sort T2, which has 250 pages. The cost is $2 * 4 * 250 = 2000$ page I/Os. To merge the sorted versions of T1 and T2, we need to scan these tables, and the cost of this step is $10 + 250 = 260$. The final projection is done on-the-fly, and by convention we ignore the cost of writing the final result.

The total cost of the plan shown in Figure 12.6 is the sum of the cost of the selection (1000+10+500+250 = 1760) and the cost of the join (40+2000+260 = 23(0), that is, 4060 page I/Os.

Sort-merge join is one of several join methods. We may be able to reduce the cost of this plan by choosing a different join method. As an alternative, suppose that we used block nested loops join instead of sort-merge join. Using T1 as the outer table, for every three-page block of T1, we scan all of T2; thus, we scan T2 four times. The cost of the join is therefore the cost of scanning T1 (10) plus the cost of scanning T2 (4 * 250 = 1000). The cost of the plan is now 1760 + 1010 = 2770 page I/Os.

A further refinement is to push the projection, just like we pushed the selections past the join. Observe that only the *sid* attribute of T1 and the *sid* and *sname* attributes of T2 are really required. As we scan Reserves and Sailors to do the selections, we could also eliminate unwanted columns. This on-the-fly projection reduces the sizes of the temporary tables T1 and T2. The reduction in the size of T1 is substantial because only an integer field is retained. In fact, T1 now fits within three buffer pages, and we can perform a block nested loops join with a single scan of T2. The cost of the join step drops to under 250 page I/Os, and the total cost of the plan drops to about 2000 I/Os.

12.5.2 Using Indexes

If indexes are available on the Reserves and Sailors tables, even better query evaluation plans may be available. For example, suppose that we have a clustered static hash index on the *bid* field of Reserves and another hash index on the *sid* field of Sailors. We can then use the query evaluation plan shown in Figure 12.7.

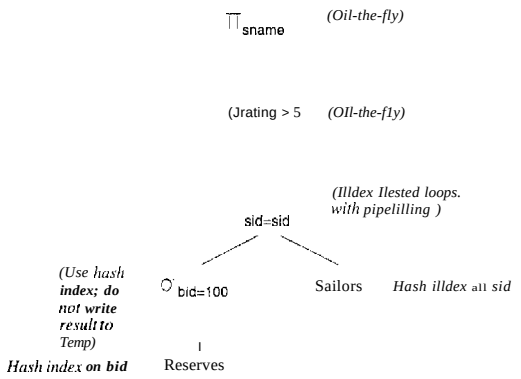


Figure 12.7 A Query Evaluation Plan Using Indexes

The selection *bid*=100 is performed on Reserves by using the hash index on *bid* to retrieve only matching tuples. As before, if we know that 100 boats are available and assume that reservations are spread out uniformly across all boats,

we can estimate the number of selected tuples to be 100, 000/100 = l000. Since the index on *b'id* is clustered, these 1000 tuples appear consecutively within the same bucket; therefore, the cost is 10 page l/Os.

:For each selected tuple, we retrieve matching Sailors tuples using the hash index on the *sid* field; selected Reserves tuples are not materialized and the join is pipelined. For each tuple in the result of the join, we perform the selection *rating*>5 and the projection of *sname* on-the-fly. There are several important points to note here:

1. Since the result of the selection on Reserves is not materialized, the optimization of projecting out fields that are not needed subsequently is unnecessary (and is not used in the plan shown in Figure 12.7).
2. The join field *sid* is a key for Sailors. Therefore, at most one Sailors tuple matches a given Reserves tuple. The cost of retrieving this matching tuple depends on whether the directory of the hash index on the *sid* column of Sailors fits in memory and on the presence of overflow pages (if any). However, the cost does *not* depend on whether this index is clustered because there is at most one matching Sailors tuple and requests for Sailors tuples are made in random order by *sid* (because Reserves tuples are retrieved by *bid* and are therefore considered in random order by *sid*). For a hash index, 1.2 page l/Os (on average) is a good estimate of the cost for retrieving a data entry. Assuming that the *sid* hash index on Sailors uses Alternative (1) for data entries, 1.2 l/Os is the cost to retrieve a matching Sailors tuple (and if one of the other two alternatives is used, the cost would be 2,2 l/Os).
3. We have chosen not to push the selection *rating*>5 ahead of the join, and there is an important reason for this decision. If we performed the selection before the join, the selection would involve scanning Sailors, assuming that no index is available on the *rating* field of Sailors. Further, whether or not such an index is available, once we apply such a selection, we have no index on the *sid* field of the result of the selection (unless we choose to build such an index solely for the sake of the subsequent join). Thus, pushing selections ahead of joins is a good heuristic, but not always the best strategy. Typically, as in this example, the existence of useful indexes is the reason a selection is *not* pushed. (Otherwise, selections are pushed.)

Let us estimate the cost of the plan shown in Figure 12.7. The selection of Reserves tuples costs 10 l/Os, as we saw earlier. There are 1000 such tuples, and for each, the cost of finding the matching Sailors tuple is 1.2 l/Os, on average. The cost of this step (the join) is therefore 1200 l/Os. All remaining selections and projections are performed on-the-fly. The total cost of the plan is 1210 l/Os.

As noted earlier, this plan does not utilize clustering of the Sailors index. The plan can be further refined if the index on the *sid* field of Sailors is clustered. Suppose we materialize the result of performing the selection *bid*=100 on Reserves and sort this temporary table. This table contains 10 pages. Selecting the tuples costs 10 page I/Os (as before), writing out the result to a temporary table costs another 10 I/Os, and with five buffer pages, sorting this temporary costs 2 * 2 * 10 = 40 I/Os. (The cost of this step is reduced if we push the projection on *sid*. The *sid* column of materialized Reserves tuples requires only three pages and can be sorted in memory with five buffer pages.) The selected Reserves tuples can now be retrieved in order by *sid*.

If a sailor has reserved the same boat many times, all corresponding Reserves tuples are now retrieved consecutively; the matching Sailors tuple will be found in the buffer pool on all but the first request for it. This improved plan also demonstrates that pipelining is not always the best strategy.

The combination of pushing selections and using indexes illustrated by this plan is very powerful. If the selected tuples from the outer table join with a single inner tuple, the join operation may become trivial, and the performance gains with respect to the naive plan in Figure 12.6 are even more dramatic. The following variant of our example query illustrates this situation:

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   Rsid = S.sid
        AND R.bid = 100 AND S.rating > 5
        AND Rday = '8/9/2002'
```

A slight variant of the plan shown in Figure 12.7, designed to answer this query, is shown in Figure 12.8. The selection *day='8/9/2002'* is applied on-the-fly to the result of the selection *bid=100* on the Reserves table.

Suppose that *bid* and *day* form a key for Reserves. (Note that this assumption differs from the schema presented earlier in this chapter.) Let us estimate the cost of the plan shown in Figure 12.8. The selection *bid=100* costs 10 page I/Os, as before, and the additional selection *day='8/9/2002'* is applied on-the-fly, eliminating all but (at most) one Reserves tuple. There is at most one rnatching Sailors tuple, and this is retrieved in 1.2 I/Os (an average value). The selection on *rating* and the projection on *sname* are then applied on-the-fly at no additional cost. The total cost of the plan in Figure 12.8 is thus about 11 I/Os. In contrast, if we modify the naive plan in Figure 12.6 to perform the additional selection on *day* together with the selection *bid=100*, the cost remains at 501,000 I/Os.

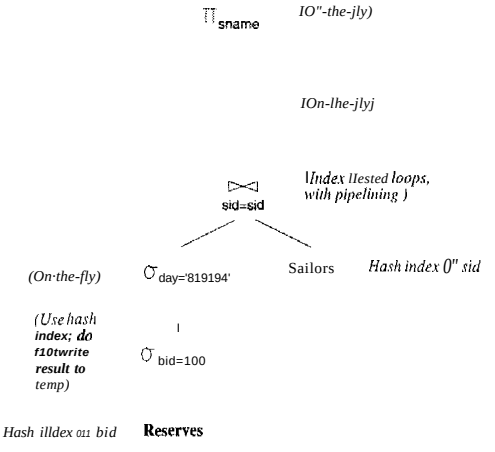


Figure 12.8 A Query Evaluation Plan for the Second Example

12.6 WHAT A TYPICAL OPTIMIZER DOES

A relational query optimizer uses relational algebra equivalences to identify many equivalent expressions for a given query. For each such equivalent version of the query, all available implementation techniques are considered for the relational operators involved, thereby generating several alternative queryevaluation plans. The optimizer estimates the cost of each such plan and chooses the one with the lowest estimated cost.

12.6.1 Alternative Plans Considered

Two relational algebra expressions over the same set of input tables are said to be **equivalent** if they produce the same result on all instances of the input tables. Relational algebra equivalences playa central role in identifying alternative plans.

Consider a basic SQL query consisting of a SELECT clause, a FROM clause, and a WHERE clause, This is easily represented as an algebra expression; the fields mentioned in the SELECT are projected from the cross-product of tables in the FROM clause, after applying the selections in the WHERE clause. The use of equivalences enable us to convert this initial representation into equivalent expressions. In particular:

- Selections and cross-products can be combined into joins.
- Joins can be extensively reordered.

- Selections and projections, which reduce the size of the input, can be "pushed" ahead of joins.

The query discussed in Section 12.5 illustrates these points; pushing the selection in that query ahead of the join yielded a dramatically better evaluation plan. \Ve discuss relational algebra equivalences in detail in Section 15.3.

Left-Deep Plans

Consider a query of the form $A \bowtie B \bowtie C \bowtie D$; that is, the natural join of four tables. Three relational algebra operator trees that are equivalent to this query (based on algebra equivalences) are shown in Figure 12.9. By convention, the left child of a join node is the outer table and the right child is the inner table. By adding details such as the join method for each join node, it is straightforward to obtain several query evaluation plans from these trees.

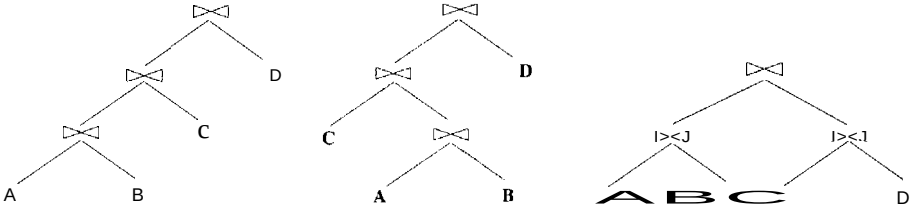


Figure 12.9 Three Join Trees

The first two trees in Figure 12.9 are examples of linear trees. In a linear tree, at least one child of a join node is a base table. The first tree is an example of a left-deep tree-the *right* child of each join node is a base table. The third tree is an example of a non-linear or bushy tree.

Optimizers typically use a dynamic-programming approach (see Section 15.4.2) to efficiently search the class of aU left-deep plans. The second and third kinds of trees are therefore never considered. Intuitively, the first tree represents a plan in which we join A and B first, then join the result with C, then join the result with D. There are 23^5 other left-deep plans that differ only in the order that tables are joined. If any of these plans has selection and projection conditions other than the joins themselves, these conditions are applied as early as possible (consitent with algebra equivalences) given the choice of a join order for the tables.

Of course, this decision rules out many alternative plans that may cost less than the best plan using a left-deep tree; we have to live with the fact that

[5]The reader should think through the number 23 in this example.