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 memory and *output* from memory to disk) dominates the cost of typical database operations, and database systems are carefully optimized to minimize 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 manager*, 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 manager 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 scans; a **scan** operation allows us to step through all the records in the file one at a time. A relation 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 available 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 $\langle k,\ rid \rangle$ 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 example, 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 index* and *secondary index* are sometimes used with a different

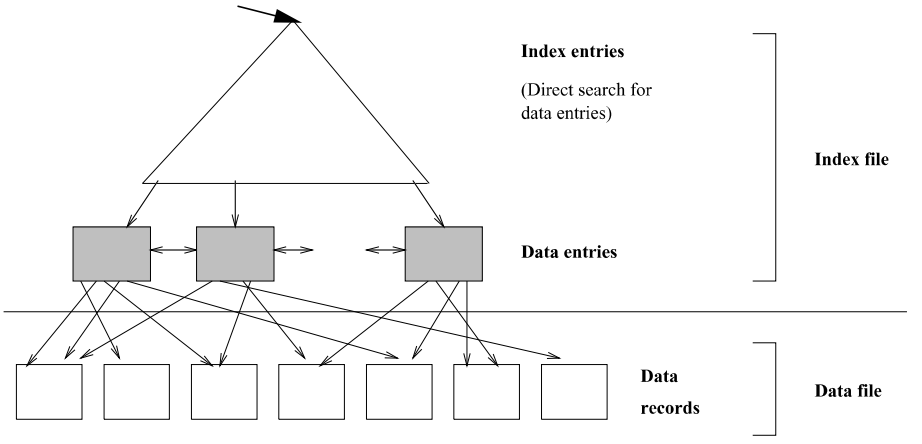

Figure 8.1 Unclustered Index Using Alternative (2)

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 that 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. We discuss this next.

8.3 INDEX DATA STRUCTURES

One way to organize data entries is to hash data entries on the search 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 in 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 I/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 I/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 $\langle sal, rid \rangle$ 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 $\langle sal,$ *rid of employee record*$\rangle$ pairs; that is, Alternative (2) is used for data entries.

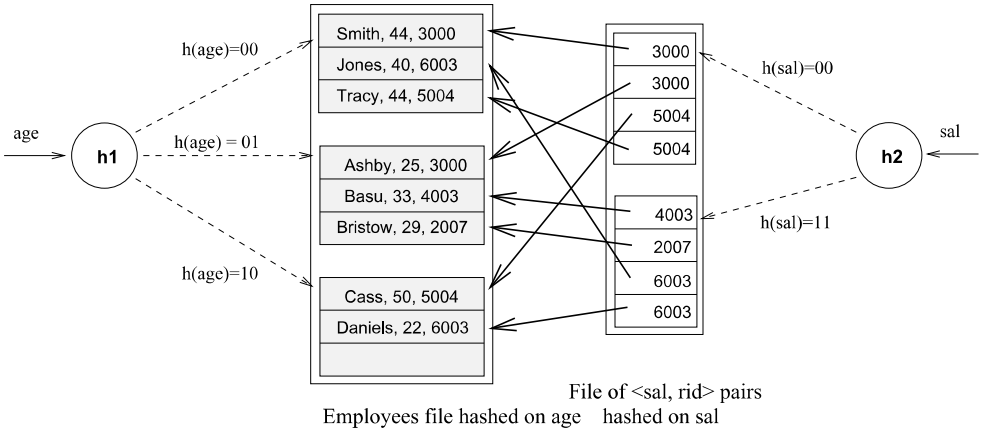

Figure 8.2 Index-Organized File Hashed on *age*, with Auxiliary Index on *sal*

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 key *age*. 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.

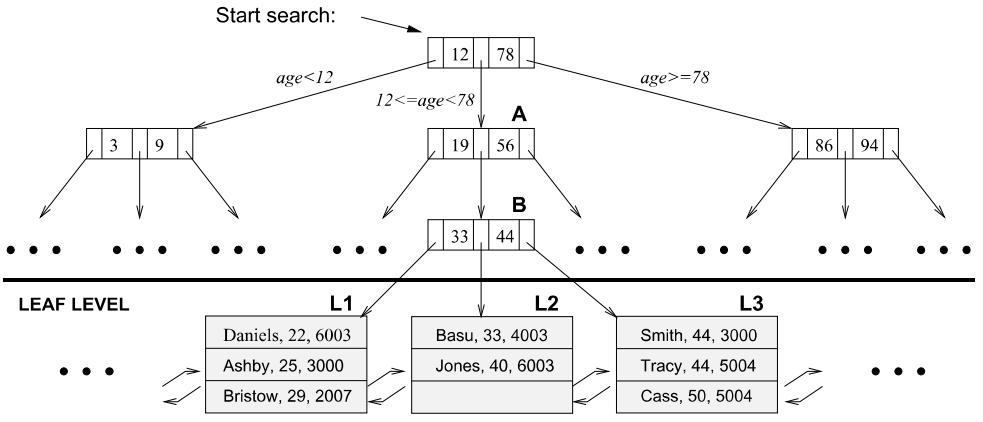

Figure 8.3 Tree-Structured Index

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 $\langle age, sal \rangle$.
- Clustered B+ tree file with search key $\langle age, sal \rangle$.
- Heap file with an unclustered B+ tree index on $\langle age, sal \rangle$.
- Heap file with an unclustered hash index on $\langle age, sal \rangle$.

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 following 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 $\langle age, sal \rangle$. 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 $log_2 B$ 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 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 $C log_2 R$. The cost is $D log_2 B + C log_2 R$, which is a significant improvement over searching heap files.

[1] 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 ($Dlog_2B+Clog_2R$) 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 $2 \cdot (0.5B(D + RC))$, 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.

[2]Unlike 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 number 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 $\langle age, sal \rangle$. We can locate the first page containing the desired record or records, should any qualifying records exist, in $log_F 1.5B$ 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 $Clog_2 R$. The cost is $Dlog_F 1.5B + Clog_2 R$, 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 ($Dlog_F 1.5B + Clog_2 R$) 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 new 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, $Dlog_F 1.5B + Clog_2 R + D$.

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 $0.1(1.5B) = 0.15B$, 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 $\langle age, sal \rangle$.

We can read all data entries at a cost of $0.15B(D + 6.7RC)$ I/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 $log_F 0.15B$ 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 qualifying 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 from the employee file with another I/O. The cost is $Dlog_F 0.15B + Clog_2 6.7R + D$, 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 of locating the first qualifying data entry ($Dlog_F 0.15B + Clog_2 6.7R$) 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 $Dlog_F 0.15B + Clog_2 6.7R$, 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 $Dlog_F 0.15B + Clog_2 6.7R + D$. 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 80 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 80 percent full, when records are initially loaded into a hashed file structure. The number of pages required to store data entries is therefore 1.25 times the number of pages when the entries are densely packed, that is, $1.25(0.10B) = 0.125B$. The number of data entries that fit on a page is $10(0.80R) = 8R$, 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 slightly 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 $0.125B(D + 8RC)$ 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 $\langle age, sal \rangle$. 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 $0.5(8R)C = 4RC$. 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 faster 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	$Dlog_2B$	$Dlog_2B$+# *matching pages*	*Search* + BD	*Search*+ BD
Clustered	$1.5BD$	$Dlog_F1.5B$	$Dlog_F1.5B$+# *matching pages*	*Search* + D	*Search*+ D
Unclustered tree index	$BD(R + 0.15)$	$D(1 + log_F0.15B)$	$D(log_F0.15B$+# *matching records*$)$	$D(3 + log_F0.15B)$	*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 much better than in a sorted file. We 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 large, 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 unclustered indexes on a data file. Suppose that employee records are sorted by *age*, or stored in a clustered file with search key *age*. If, in addition, we have an index on the *sal* field, the latter must be an unclustered index. We can also build an unclustered 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 some 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 on a range query, consider the following example:

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

If we have a B+ 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? If virtually everyone is older than 40, we gain little by using an index on *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    E.dno, 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 *dno*, 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 *dno* might suit our purposes better. We could use the index to retrieve all tuples, grouped by *dno*, and for each *dno* 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 *dno*.) 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 *dno* 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 *dno*. Therefore, this query suggests that we build a clustered index on *dno* 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 $\langle age, sal \rangle$, a composite index with key $\langle sal, age \rangle$, 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 $sal > 40$.

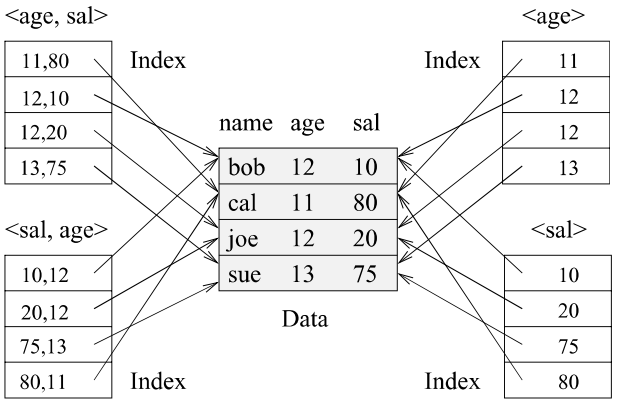

Figure 8.5 Composite Key Indexes

Note 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 satisfy the condition. For selections of the form $condition \wedge \ldots \wedge condition$, we can define when an index matches the selection as follows:[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, $\langle age \rangle$ and $\langle age, sal, department \rangle$ are prefixes of key $\langle age, sal, department \rangle$, but $\langle age, department \rangle$ and $\langle sal, department \rangle$ 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 compression 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 $\langle age, sal \rangle$ 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 $\langle age, sal \rangle$ is as effective as a composite, clustered B+ tree index on $\langle sal, age \rangle$. 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 $\langle age, sal \rangle$ 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 $\langle sal, age \rangle$ 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. We 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 E.sal BETWEEN 3000 AND 5000
```

A composite B+ tree index on $\langle age, sal \rangle$ allows us to answer the query with an index-only scan. A composite B+ tree index on $\langle sal, age \rangle$ 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 $\langle age, sal \rangle$.

Here is a variation of an earlier example:

```
SELECT    E.dno, COUNT(*)
FROM      Employees E
WHERE     E.sal=10,000
GROUP BY E.dno
```

An index on *dno* 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 $\langle sal, dno \rangle$ or $\langle dno, sal \rangle$. In an index with key $\langle sal, dno \rangle$, all data entries with $sal = 10,000$ are arranged contiguously (whether or not the index is clustered). Further, these entries are sorted by *dno*, making it easy to obtain a count for each *dno* 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 *dno* to identify the groups (because, for example, two entries with the same *dno* but different *sal* values may not be contiguous). An index with key $\langle dno, sal \rangle$ is better for this query: Data entries with a given *dno* value are stored together, and each such group of entries is itself sorted by *sal*. For each *dno* 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 $\langle sal, dno \rangle$ 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 *dno*:

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

An index on *dno* 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 *dno* value are stored together (whether or not the index is clustered). Further, this group of entries is itself sorted by *sal*. An index on $\langle sal, dno \rangle$ 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, the 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 $\langle age, gpa \rangle$, 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? When 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 unclustered 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 decisions 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 *index-only* evaluation method? What is its primary advantage? **(Section 8.5.2)**
- What is a *composite search 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. Why 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. What 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 are supported by the file of records abstraction?
2. What is an index on a file of records? What is a search key for an index? Why do we need indexes?

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	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? What 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? Would 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 an index cannot be constructed, say so and explain why.

1. An unclustered index on *age* using Alternative (1).
2. An unclustered 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 Alternative (2).
6. A clustered index on *age* using Alternative (3).
7. An unclustered 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 does 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 *eid* and an unclustered 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 $\langle ename, age, sal \rangle$ fields of Emp.
   (b) Unclustered hash index on $\langle ename, age, sal \rangle$ fields of Emp.
   (c) Clustered B+ tree index on $\langle ename, age, sal \rangle$ fields of Emp.
   (d) Unclustered hash index on $\langle eid, did \rangle$ 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 $\langle floor, budget \rangle$ 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 completely unhappen.

—Edward DeBono

This chapter initiates a study of the internals of an RDBMS. 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 *files 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

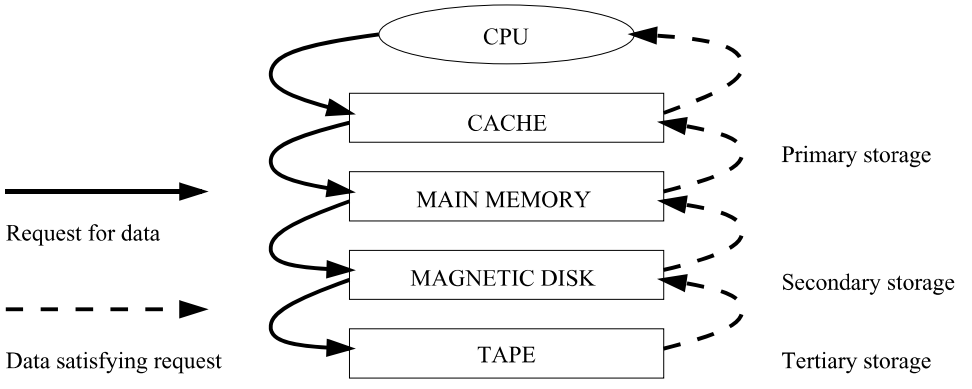

Figure 9.1 The Memory Hierarchy

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 memory 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, consider 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. Blocks 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

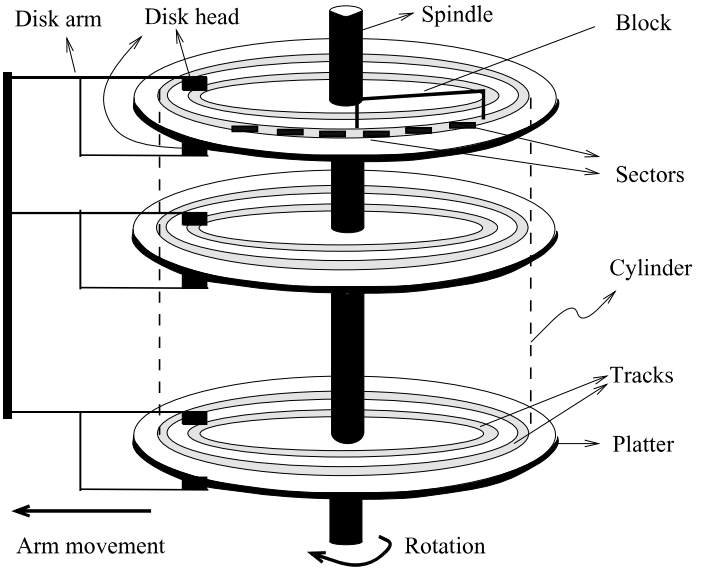

Figure 9.2 Structure of a Disk

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 head. 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 on 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 memory 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:

$$\textit{access time} = \textit{seek time} + \textit{rotational delay} + \textit{transfer time}$$

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 dominates 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 I/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 example, 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 I 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 any one 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 any one 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 10-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 *redundancy 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 does not have the best read performance 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 0 solution consists of only four data disks. Independent of the number of data disks, the effective space utilization for a RAID Level 0 system is always 100 percent.

Level 1: 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 maintained. 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 Level 1 implementation. The effective space utilization is 50 percent, independent of the number of data disks.

Level 0+1: Striping and Mirroring

RAID Level 0+1—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 from the aggregated bandwidth of all disks. The cost for writes is analogous to RAID Level 1.

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 write 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:

$$\texttt{NewParity} = (\texttt{OldData} \text{ XOR } \texttt{NewData}) \text{ XOR } \texttt{OldParity}$$

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 and large 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 space 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 might 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 use, 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 OS 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 be built using OS files. For example, the entire database could reside in one or more OS files for which a number of blocks are allocated (by the OS) 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 OS file system and instead do their own disk management, either from scratch or by extending OS facilities. The reasons are practical as well as technical. One practical reason is that a DBMS vendor who wishes to support several OS platforms 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 system, the largest file size is 4 GB, whereas a DBMS may want to access a single file

larger than that. A related problem is that typical OS 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 OS 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: *pin_count* 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.

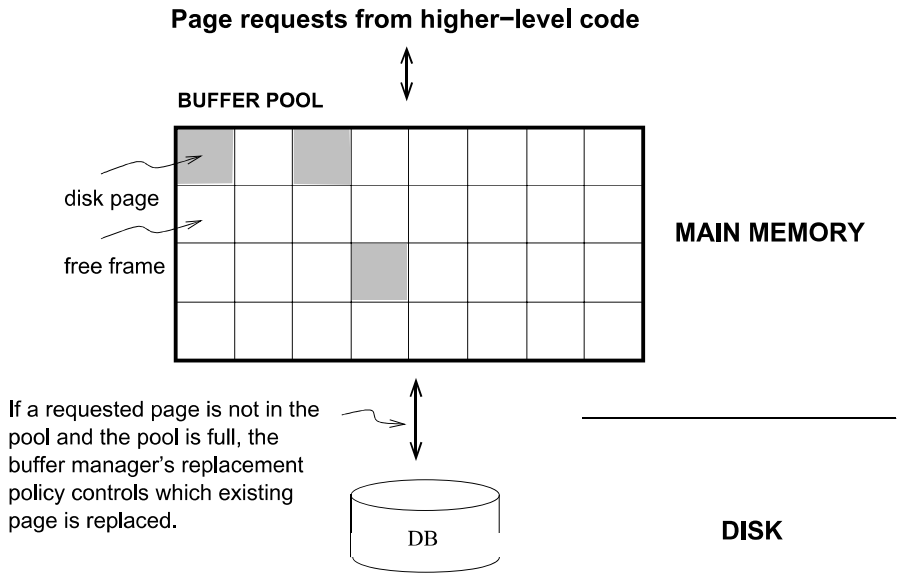

Figure 9.3 The Buffer Pool

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 modify. 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 manager simply assumes that 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 many 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**, among 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 OS 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 OS 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 operators) 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 and list prefetch (prefetching a list of pages). In general, the prefetch size is 32 4KB pages, but this can be set by the user. For some sequential type database utilities (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 of a 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 prefetched 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 system crashes in the interim, the effects can be catastrophic for a DBMS. (Crash recovery is discussed 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 direct 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 *create* 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. We discuss two alternative ways to maintain 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 $\langle heap\_file\_name, page\_1\_addr \rangle$ 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.

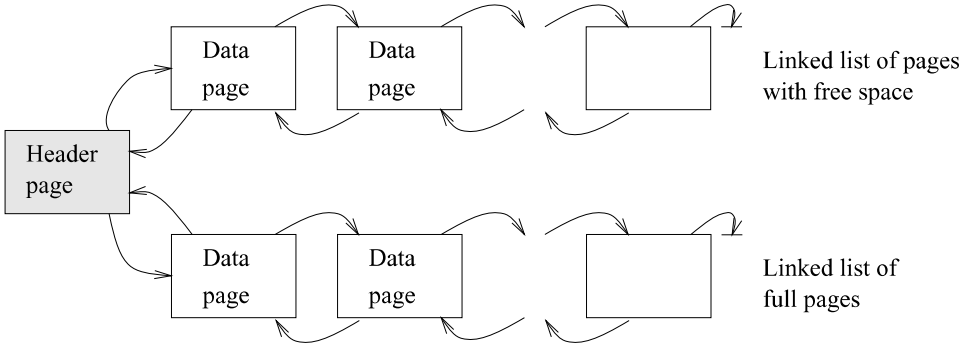

Figure 9.4 Heap File Organization with a Linked List

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 manager 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 scheme 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 examine 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.)

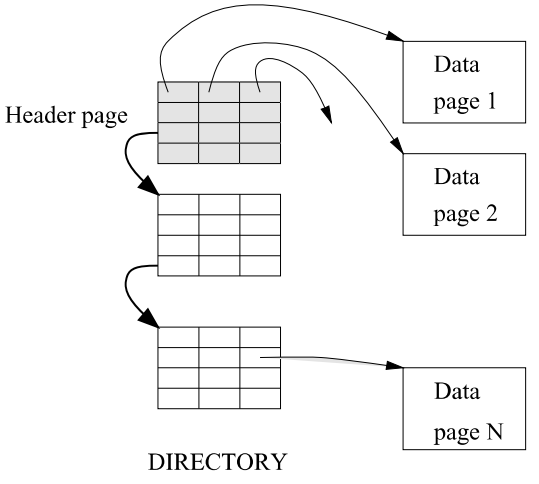

Figure 9.5 Heap File Organization with a Directory

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 Commercial 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 $\langle$*page id*, *slot number*$\rangle$; 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 $\langle$*page id*, *slot number*$\rangle$ 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 are 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 does 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.

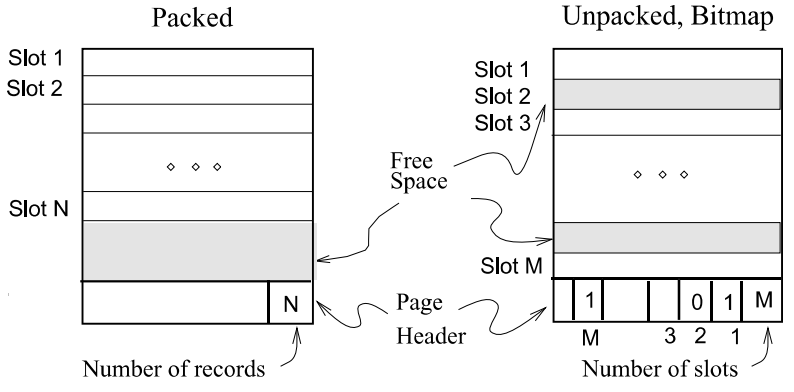

Figure 9.6 Alternative Page Organizations for Fixed-Length Records

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, and 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.

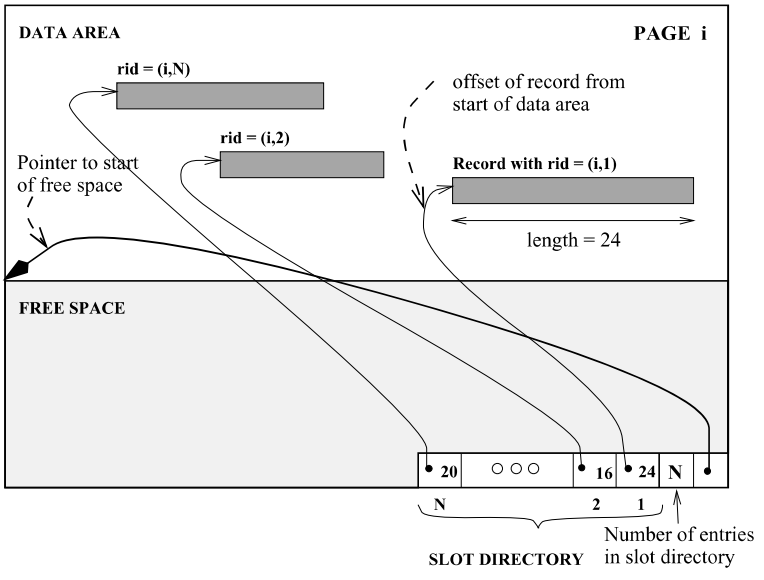

Figure 9.7 Page Organization for Variable-Length Records

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 reclaim 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 may 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 same 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 formats, we note that in addition to storing individual records, information common 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.

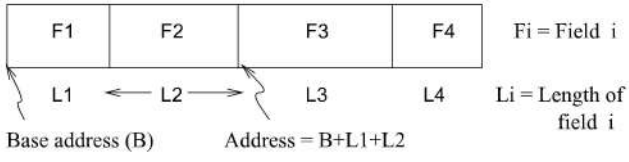

Figure 9.8 Organization of Records with Fixed-Length Fields

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 organization 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.

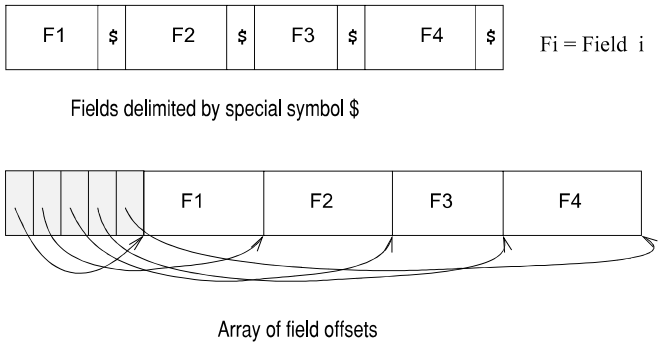

Figure 9.9 Alternative Record Organizations for Variable-Length Fields

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 *null* values and adding fields to a record type arise with fixed-length records as well.

Having 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 presented.
- 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 rids, 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 new location of the record. And to ensure that space is always 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 *any* one 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 system is and how it improves performance and reliability. Discuss *striping* and its impact on performance and *redundancy* and its impact 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 pool*? 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 enable us to move records on a page without altering the record's identifier? What are 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 many 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 time is required to read a file containing 100,000 records of 100 bytes each sequentially? Again, how would 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 record 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 must do to process a read request for a page. What 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? Who is responsible for unpinning pages?

Exercise 9.10 When a page in the buffer 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 What happens if a page is requested when all pages in the buffer pool are dirty?

Exercise 9.12 What 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 organization 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 organization 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*, and the papers contain much of interest from that standpoint as well. Other papers that cover storage management issues in the context of significant implemented prototype systems are [480] and [588]. The Dali storage manager, which is optimized for main memory databases, is described in [406]. Three techniques for implementing long fields are compared in [96]. The impact of processor cache misses on DBMS performance 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, 261, 235].