8 FILE ORGANIZATIONS & INDEXES

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

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

A **file organization** is a way of arranging the records in a file when the file is stored on disk. A file of records is likely to be accessed and modified in a variety of ways, and different ways of arranging the records enable different operations over the file to be carried out efficiently. For example, if we want to retrieve employee records in alphabetical order, sorting the file by name is a good file organization. On the other hand, if we want to retrieve all employees whose salary is in a given range, sorting employee records by name is not a good file organization. A DBMS supports several file organization techniques, and an important task of a DBA is to choose a good organization for each file, based on its expected pattern of use.

We begin this chapter with a discussion in Section 8.1 of the cost model that we use in this book. In Section 8.2, we present a simplified analysis of three basic file organizations: *files of randomly ordered records* (i.e., heap files), *files sorted on some field*, and *files that are hashed on some fields*. Our objective is to emphasize the importance of choosing an appropriate file organization.

Each file organization makes certain operations efficient, but often we are interested in supporting more than one operation. For example, sorting a file of employee records on the *name* field makes it easy to retrieve employees in alphabetical order, but we may also want to retrieve all employees who are 55 years old; for this, we would have to scan the entire file. To deal with such situations, a DBMS builds an index, as we described in Section 7.5.2. An index on a file is designed to speed up operations that are not efficiently supported by the basic organization of records in that file. Later chapters cover several specific index data structures; in this chapter we focus on properties of indexes that do not depend on the specific index data structure used.

Section 8.3 introduces indexing as a general technique that can speed up retrieval of records with given values in the search field. Section 8.4 discusses some important properties of indexes, and Section 8.5 discusses DBMS commands to create an index.

8.1 COST MODEL

In this section we introduce a cost model that allows us to estimate the cost (in terms of execution time) of different database operations. We will use the following notation and assumptions in our analysis. There are B data pages with R 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 will 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.

Typical values today are $D = 15$ milliseconds, C and $H = 100$ nanoseconds; we therefore expect the cost of I/O to dominate. This conclusion is supported by current hardware trends, in which 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 therefore use the number of disk page I/Os as our cost metric in this book.

- We emphasize that real systems must consider other aspects of cost, such as CPU costs (and transmission costs in a distributed database). However, our goal is primarily to present the underlying algorithms and to illustrate how costs can be estimated. Therefore, for simplicity, we have chosen to concentrate on only the I/O component of cost. Given the fact that I/O is often (even typically) the dominant component of the cost of database operations, considering I/O costs gives us a good first approximation to the true costs.

- 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 have therefore chosen to use a simplistic model in which we just count the number of pages that are read from or written to disk as a measure of I/O. We have ignored the important issue of **blocked access**—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 to **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: We would have an additional seek cost for each page in the block.

This discussion of the cost metric we have chosen must be kept in mind when we discuss the cost of various algorithms in this chapter and in later chapters. We discuss the implications of the cost model whenever our simplifying assumptions are likely to affect the conclusions drawn from our analysis in an important way.

8.2 COMPARISON OF THREE FILE ORGANIZATIONS

We now compare the costs of some simple operations for three basic file organizations: *files of randomly ordered records, or heap files*; *files sorted on a sequence of fields*; and *files that are hashed on a sequence of fields.* For sorted and hashed files, the sequence of fields (e.g., *salary*, *age*) on which the file is sorted or hashed is called the **search key**. Note that the search key for an index can be any sequence of one or more fields; it need not uniquely identify records. We observe that 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.

Our goal is to emphasize how important the choice of an appropriate file organization can be. The operations that we consider are described below.

- **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 Students record for the student with *sid* 23." 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 Students records with *name* alphabetically after 'Smith.' "
- **Insert:** 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:** 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.

8.2.1 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 there is no record that 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 students aged 18"), we always have to scan the entire file because several 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.

8.2.2 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.

Search with equality selection: We assume that the equality selection is specified on the field by which the file is sorted; if not, the cost is identical to that for a heap

[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 7. This increases the cost of insertion and deletion a little, but not enough to affect our comparison of heap files, sorted files, and hashed files.

file. We can locate the first page containing the desired record or records, should any qualifying records exist, with a binary search in log_2B 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. This assumption is not valid if, for example, the sorted file is implemented as a heap file using the linked-list organization, with pages in the appropriate sorted order.) 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 $Clog_2R$. The cost is $Dlog_2B + Clog_2R$, which is a significant improvement over searching heap files.

If there are several qualifying records (e.g., "Find all students 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 there are no qualifying records, 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 is on the sort field, the first record that satisfies the selection is located as it is 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 will be 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. Thus, we must read the latter half of the file and then write it back after adding the new record. The cost is therefore the cost of searching to find the position of the new record plus $2 * (0.5B(D + RC))$, that is, search cost plus $B(D + RC)$.

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

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

8.2.3 Hashed Files

A simple hashed file organization enables us to locate records with a given search key value quickly, for example, "Find the Students record for Joe," if the file is hashed on the *name* field.

The pages in a hashed file are grouped into **buckets**. Given a bucket number, the hashed file structure allows us to find the **primary page** for that bucket. The bucket to which a record belongs can be determined by applying a special function called a **hash function**, to the search field(s). On inserts, a record is inserted into the appropriate bucket, with additional 'overflow' pages allocated if the primary page for the bucket becomes full. The overflow pages for each bucket are maintained in a linked list. To search for a record with a given search key value, we simply apply the hash function to identify the bucket to which such records belong and look at all pages in that bucket.

This organization is called a **static hashed file**, and its main drawback is that long chains of overflow pages can develop. This can affect performance because all pages in a bucket have to be searched. Dynamic hash structures that address this problem are known, and we discuss them in Chapter 10; for the analysis in this chapter, we will simply assume that there are no overflow pages.

Scan: In a hashed file, pages are kept at about 80 percent occupancy (to leave some space for future insertions and minimize overflow pages 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 organized into a hashed file structure. Thus, the number of pages, and the cost of scanning all the data pages, is about 1.25 times the cost of scanning an unordered file, that is, $1.25B(D + RC)$.

Search with equality selection: This operation is supported very efficiently if the selection is on the search key for the hashed file. (Otherwise, the entire file must

[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.

be scanned.) The cost of identifying the page that contains qualifying records is H; assuming that this bucket consists of just one page (i.e., no overflow pages), retrieving it costs D. The cost is $H + D + 0.5RC$ if we assume that we find the record after scanning half the records on the page. This is even lower than the cost for sorted files. If there are several qualifying records, or none, we still have to retrieve just one page, but we must scan the entire page.

Note that the hash function associated with a hashed file maps a record to a bucket based on the values in *all* the search key fields; if the value for any one of these fields is not specified, we cannot tell which bucket the record belongs to. Thus, if the selection is not an equality condition on all the search key fields, we have to scan the entire file.

Search with range selection: The hash structure offers no help; even if the range selection is on the search key, the entire file must be scanned. The cost is $1.25B(D + RC)$.

Insert: The appropriate page must be located, modified, and then written back. The cost is the cost of search plus $C + D$.

Delete: We must search for the record, remove it from the page, and write the modified page back. The cost is again the cost of search plus $C + D$ (for writing the modified page).

If records to be deleted are specified using an equality condition on the search key, all qualifying records are guaranteed to be in the same bucket, which can be identified by applying the hash function.

8.2.4 Choosing a File Organization

Figure 8.1 compares I/O costs for the three file organizations. A heap file has good storage efficiency and supports fast scan, insertion, and deletion of records. However, it is slow for searches.

File Type	*Scan*	*Equality Search*	*Range Search*	*Insert*	*Delete*
Heap	BD	$0.5BD$	BD	$2D$	$Search + D$
Sorted	BD	$Dlog_2B$	$Dlog_2B + \#\ matches$	$Search + BD$	$Search + BD$
Hashed	$1.25BD$	D	$1.25BD$	$2D$	$Search + D$

Figure 8.1 A Comparison of I/O Costs

A sorted file also offers good storage efficiency, but insertion and deletion of records is slow. It is quite fast for searches, and it is the best structure for range selections. It is worth noting that in a real DBMS, a file is almost never kept fully sorted. A structure called a B+ tree, which we will discuss in Chapter 9, 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.)

Files are sometimes kept 'almost sorted' in that they are originally sorted, with some free space left on each page to accommodate future insertions, but once this space is used, overflow pages are used to handle insertions. The cost of insertion and deletion is similar to a heap file, but the degree of sorting deteriorates as the file grows.

A hashed file does not utilize space quite as well as a sorted file, but insertions and deletions are fast, and equality selections are very fast. However, the structure offers no support for range selections, and full file scans are a little slower; the lower space utilization means that files contain more pages.

In summary, Figure 8.1 demonstrates that no one file organization is uniformly superior in all situations. An unordered file is best if only full file scans are desired. A hashed file is best if the most common operation is an equality selection. A sorted file is best if range selections are desired. The organizations that we have studied here can be improved on—the problems of overflow pages in static hashing can be overcome by using dynamic hashing structures, and the high cost of inserts and deletes in a sorted file can be overcome by using tree-structured indexes—but the main observation, that the choice of an appropriate file organization depends on how the file is commonly used, remains valid.

8.3 OVERVIEW OF INDEXES

As we noted earlier, an index on a file is an auxiliary structure designed to speed up operations that are not efficiently supported by the basic organization of records in that file.

An index can be viewed as a collection of **data entries**, with an efficient way to locate all data entries with search key value k. Each such data entry, which we denote as $k*$, contains enough information to enable us to retrieve (one or more) data records with search key value k. (Note that a data entry is, in general, different from a data record!) Figure 8.2 shows an index with search key *sal* that contains $\langle sal, rid \rangle$ pairs as data entries. The *rid* component of a data entry in this index is a pointer to a record with search key value *sal*.

Two important questions to consider are:

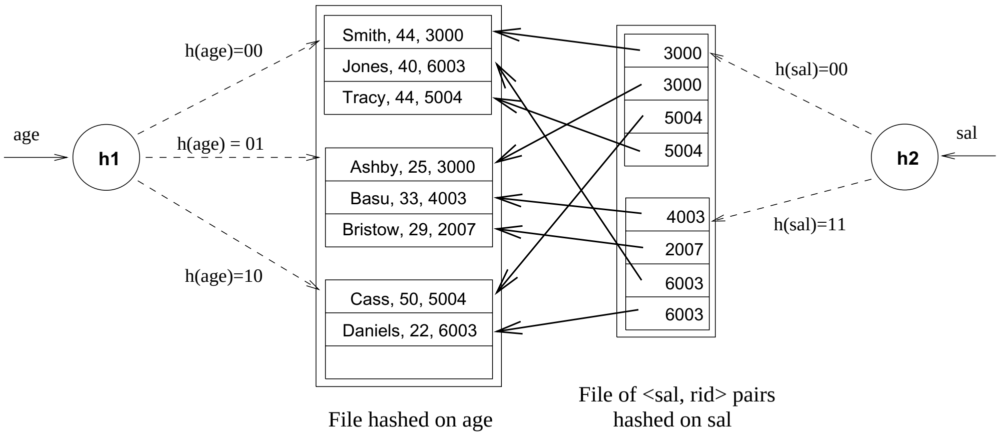

Figure 8.2 File Hashed on *age*, with Index on *sal*

- How are data entries organized in order to support efficient retrieval of data entries with a given search key value?
- Exactly what is stored as a data entry?

One way to organize data entries is to hash data entries on the search key. In this approach, we essentially treat the collection of data entries as a file of records, hashed on the search key. This is how the index on *sal* shown in Figure 8.2 is organized. 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. Another way to organize data entries is to build a data structure that directs a search for data entries. Several index data structures are known that allow us to efficiently find data entries with a given search key value. We will study tree-based index structures in Chapter 9 and hash-based index structures in Chapter 10.

We consider what is stored in a data entry in the following section.

8.3.1 Alternatives for Data Entries in an Index

A data entry $k*$ allows us to retrieve one or more data records with key value k. We need to consider three main alternatives:

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.

Observe that if an index uses Alternative (1), there is no need to store the data records separately, in addition to the contents of the index. We can think of such an index as a special file organization that can be used instead of a sorted file or a heap file organization. Figure 8.2 illustrates Alternatives (1) and (2). The file of employee records is hashed on *age*; we can think of this as an index structure in which a hash function is applied to the *age* value to locate the bucket for a record and Alternative (1) is used for data entries. The index on *sal* also uses hashing to locate data entries, which are now ⟨*sal, rid of employee record*⟩ pairs; that is, Alternative (2) is used for data entries.

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 as illustrated in Figure 8.2, at most one of the indexes should use Alternative (1) because we want to avoid storing data records multiple times.

We note that different index data structures used to speed up searches for data entries with a given search key can be combined with any of the three alternatives for data entries.

8.4 PROPERTIES OF INDEXES

In this section, we discuss some important properties of an index that affect the efficiency of searches using the index.

8.4.1 Clustered versus Unclustered 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**. An index that uses Alternative (1) is clustered, by definition. An index that uses Alternative (2) or Alternative (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. (Indexes based on hashing do not store data entries in sorted order by search key, so a hash index is clustered only if it uses Alternative (1).)

Indexes that maintain data entries in sorted order by search key use a collection of *index entries*, organized into a tree structure, to guide searches for data entries, which are stored at the leaf level of the tree in sorted order. Clustered and unclustered tree indexes are illustrated in Figures 8.3 and 8.4; we discuss tree-structured indexes further in Chapter 9. For simplicity, in Figure 8.3 we assume that the underlying file of data records is fully sorted.

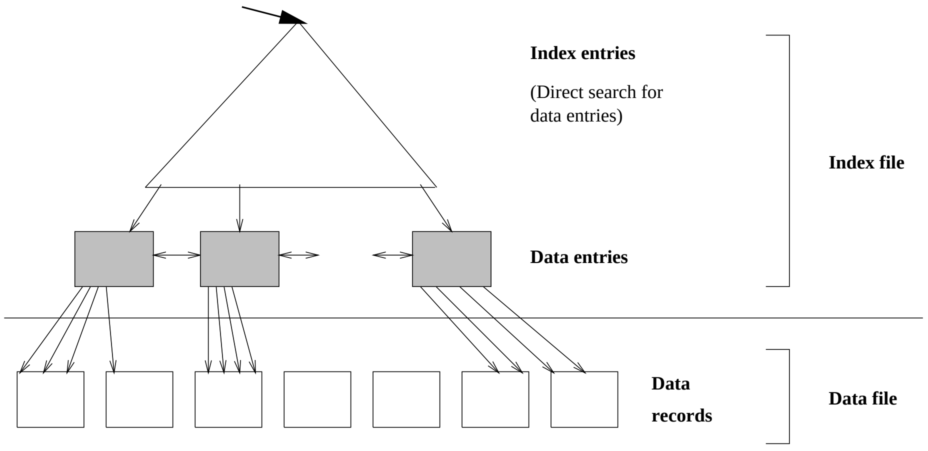

Figure 8.3 Clustered Tree Index Using Alternative (2)

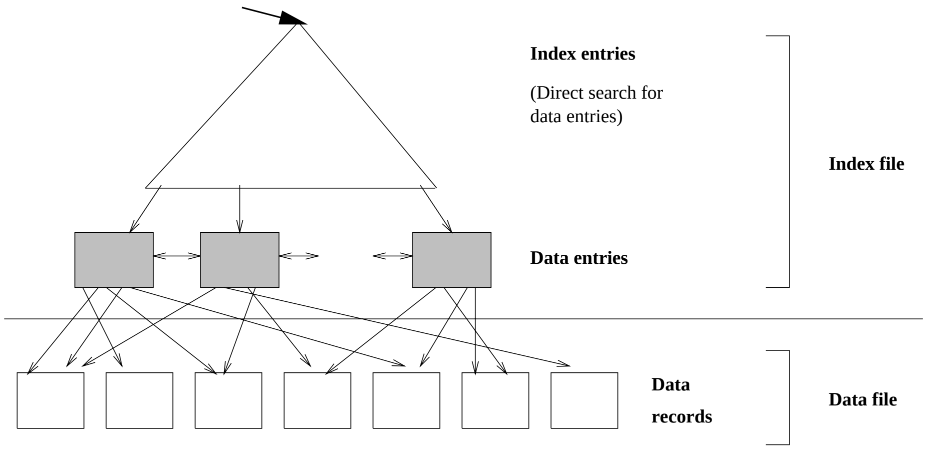

Figure 8.4 Unclustered Tree Index Using Alternative (2)

In practice, data records are rarely maintained in fully sorted order, unless data records are stored in an index using Alternative (1), because of the high overhead of moving data records around to preserve the sort order as records are inserted and deleted. Typically, the records are sorted initially and each page is left with some free space to absorb future insertions. If the free space on a page is subsequently used up (by records inserted after the initial sorting step), further insertions to this page are handled using a linked list of overflow pages. Thus, after a while, the order of records only approximates the intended sorted order, and the file must be **reorganized** (i.e., sorted afresh) to ensure good performance.

Thus, clustered indexes are relatively expensive to maintain when the file is updated. Another reason clustered indexes are expensive to maintain is that data entries may have to be moved across pages, and if records are identified by a combination of page id and slot, as is often the case, 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; these additional updates can be very time-consuming.

A data file can be clustered on at most one search key, which means that we can have at most one clustered index on a data file. An index that is not clustered is called an **unclustered** index; we can have several unclustered indexes on a data file. Suppose that Students records are sorted by *age*; an index on *age* that stores data entries in sorted order by *age* is a clustered index. If in addition we have an index on the *gpa* field, the latter must be an unclustered index.

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, the rids in qualifying data entries point to a contiguous collection of records, as Figure 8.3 illustrates, 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! This point is discussed further in Chapters 11 and 16.

8.4.2 Dense versus Sparse Indexes

An index is said to be **dense** if it contains (at least) one data entry for every search key value that appears in a record in the indexed file.[3] A **sparse** index contains one entry for each page of records in the data file. Alternative (1) for data entries always leads to a dense index. Alternative (2) can be used to build either dense or sparse indexes. Alternative (3) is typically only used to build a dense index.

We illustrate sparse and dense indexes in Figure 8.5. A data file of records with three fields (*name*, *age*, and *sal*) is shown with two simple indexes on it, both of which use Alternative (2) for data entry format. The first index is a sparse, clustered index on *name*. Notice how the order of data entries in the index corresponds to the order of

[3]We say 'at least' because several data entries could have the same search key value if there are duplicates and we use Alternative (2).

records in the data file. There is one data entry per page of data records. The second index is a dense, unclustered index on the *age* field. Notice that the order of data entries in the index differs from the order of data records. There is one data entry in the index per record in the data file (because we use Alternative (2)).

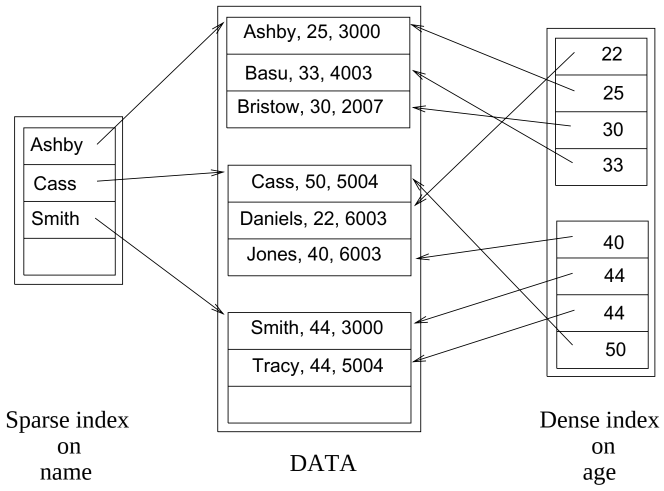

Figure 8.5 Sparse versus Dense Indexes

We cannot build a sparse index that is not clustered. Thus, we can have at most one sparse index. A sparse index is typically much smaller than a dense index. On the other hand, some very useful optimization techniques rely on an index being dense (Chapter 16).

A data file is said to be **inverted** on a field if there is a dense secondary index on this field. A **fully inverted** file is one in which there is a dense secondary index on each field that does not appear in the primary key.[4]

8.4.3 Primary and Secondary Indexes

An index on a set of fields that includes the *primary key* is called a **primary index**. An index that is not a primary index is called a **secondary** index. (The terms *primary index* and *secondary index* are sometimes used with a different 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.)

[4]This terminology arises from the observation that these index structures allow us to take the value in a non key field and get the values in key fields, which is the inverse of the more intuitive case in which we use the values of the key fields to locate the record.

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. Thus, 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.

8.4.4 Indexes Using 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 8.6 illustrates the difference between a composite index with key ⟨*age, sal*⟩, a composite index with key ⟨*sal, age*⟩, an index with key *age*, and an index with key *sal*. All indexes shown in the figure use Alternative (2) for data entries.

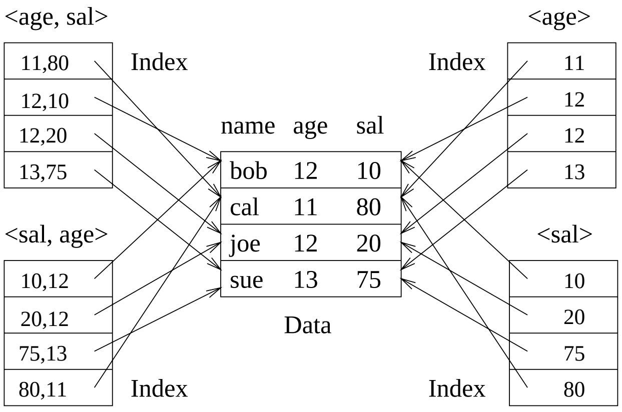

Figure 8.6 Composite Key Indexes

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.

A **range query** is one in which 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 $sal > 40$.

8.5 INDEX SPECIFICATION IN SQL-92

The SQL-92 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 9—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 the index is created, it is automatically maintained by the DBMS adding/removing data entries in response to inserts/deletes of records on the Students relation.

8.6 POINTS TO REVIEW

- A *file organization* is a way of arranging records in a file. In our discussion of different file organizations, we use a simple cost model that uses the number of disk page I/Os as the cost metric. **(Section 8.1)**

- We compare three basic file organizations (*heap files, sorted files*, and *hashed files*) using the following operations: scan, equality search, range search, insert, and delete. The choice of file organization can have a significant impact on performance. **(Section 8.2)**

- An *index* is a data structure that speeds up certain operations on a file. The operations involve a *search key*, which is a set of record fields (in most cases a single field). The elements of an index are called *data entries.* Data entries can be actual data records, ⟨*search-key, rid*⟩ pairs, or ⟨*search-key, rid-list*⟩ pairs. A given file of data records can have several indexes, each with a different search key. **(Section 8.3)**

- In a *clustered* index, the order of records in the file matches the order of data entries in the index. An index is called *dense* if there is at least one data entry per search key that appears in the file; otherwise the index is called *sparse.* An index is called a *primary index* if the search key includes the primary key; otherwise it is called a *secondary index.* If a search key contains several fields it is called a *composite key.* **(Section 8.4)**

- SQL-92 does not include statements for management of index structures, and so there some variation in index-related commands across different DBMSs. **(Section 8.5)**

EXERCISES

Exercise 8.1 What are the main conclusions that you can draw from the discussion of the three file organizations?

Exercise 8.2 Consider a delete specified using an equality condition. What is the cost if no record qualifies? What is the cost if the condition is not on a key?

Exercise 8.3 Which of the three basic file 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 8.4 Explain the difference between each of the following:

1. Primary versus secondary indexes.
2. Dense versus sparse indexes.
3. Clustered versus unclustered indexes.

If you were about to create an index on a relation, what considerations would guide your choice with respect to each pair of properties listed above?

Exercise 8.5 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 8.6 If an index contains data records as 'data entries', is it clustered or unclustered? Dense or sparse?

Exercise 8.7 Consider Alternatives (1), (2) and (3) for 'data entries' in an index, as discussed in Section 8.3.1. Are they all suitable for secondary indexes? Explain.

Exercise 8.8 Consider the instance of the Students relation shown in Figure 8.7, 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 in page 1, slot 1; the second tuple is in page 1, slot 2; and so on. Each page can store up to three data records. You can use $\langle$*page-id, slot*$\rangle$ to identify a tuple.

List the data entries in each of the following indexes. 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. A dense index on *age* using Alternative (1).
2. A dense index on *age* using Alternative (2).
3. A dense index on *age* using Alternative (3).
4. A sparse index on *age* using Alternative (1).

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 8.7 An Instance of the Students Relation, Sorted by *age*

5. A sparse index on *age* using Alternative (2).
6. A sparse index on *age* using Alternative (3).
7. A dense index on *gpa* using Alternative (1).
8. A dense index on *gpa* using Alternative (2).
9. A dense index on *gpa* using Alternative (3).
10. A sparse index on *gpa* using Alternative (1).
11. A sparse index on *gpa* using Alternative (2).
12. A sparse index on *gpa* using Alternative (3).

PROJECT-BASED EXERCISES

Exercise 8.9 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? Are sparse indexes supported?

BIBLIOGRAPHIC NOTES

Several books discuss file organizations in detail [25, 266, 381, 461, 564, 606, 680].