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:
- 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 \( \langle k, \text{rid} \rangle \) pair, where rid is the record id of a data record with search key value k.
3. A data entry is a \( \langle k, \text{rid-list} \rangle \) 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)](image)

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. 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

---

3We 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)).

![Sparse versus Dense Indexes](image)

**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 \(\langle age, sal\rangle\), 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.

![Figure 8.6 Composite Key Indexes](image)

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`. 