Index Structures

It is not sufficient simply to scatter the records that represent tuples of a relation among various blocks. To see why, think how we would answer the simple query `SELECT * FROM R`. We would have to examine every block in the storage system to find the tuples of R. A better idea is to reserve some blocks, perhaps several whole cylinders, for R. Now, at least we can find the tuples of R without scanning the entire data store.

However, this organization offers little help for a query like

```
SELECT * FROM R WHERE a=10;
```

You may recall the importance of creating *indexes* to speed up queries that specify values for one or more attributes. As suggested in Fig. 1, an index is any data structure that takes the value of one or more fields and finds the records with that value "quickly." In particular, an index lets us find a record without having to look at more than a small fraction of all possible records. The field(s) on whose values the index is based is called the *search key*, or just "key" if the index is understood.

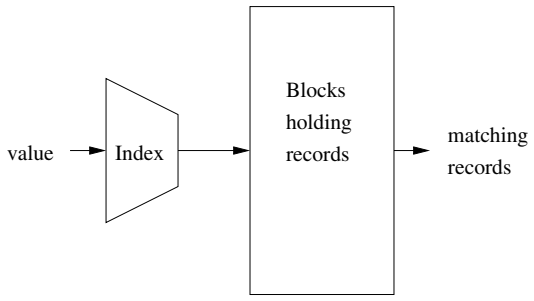

Figure 1: An index takes a value for some field(s) and finds records with the matching value

Different Kinds of "Keys"

There are many meanings of the term "key." It can be used to mean the primary key of a relation. We shall also speak of "sort keys," the attribute(s) on which a file of records is sorted. We just introduced "search keys," the attribute(s) for which we are given values and asked to search, through an index, for tuples with matching values. We try to use the appropriate adjective — "primary," "sort," or "search" — when the meaning of "key" is unclear. However, in many cases, the three kinds of keys are one and the same.

In this chapter, we shall introduce the most common form of index in database systems: the B-tree. We shall also discuss hash tables in secondary storage, which is another important index structure. Finally, we consider other index structures that are designed to handle multidimensional data. These structures support queries that specify values or ranges for several attributes at once.

1 Index-Structure Basics

In this section, we introduce concepts that apply to all index structures. Storage structures consist of *files*, which are similar to the files used by operating systems. A *data file* may be used to store a relation, for example. The data file may have one or more *index files*. Each index file associates values of the search key with pointers to data-file records that have that value for the attribute(s) of the search key.

Indexes can be "dense," meaning there is an entry in the index file for every record of the data file. They can be "sparse," meaning that only some of the data records are represented in the index, often one index entry per block of the data file. Indexes can also be "primary" or "secondary." A primary index determines the location of the records of the data file, while a secondary index does not. For example, it is common to create a primary index on the primary key of a relation and to create secondary indexes on some of the other attributes.

We conclude the section with a study of information retrieval from documents. The ideas of the section are combined to yield "inverted indexes," which enable efficient retrieval of documents that contain one or more given keywords. This technique is essential for answering search queries on the Web, for instance.

1.1 Sequential Files

A *sequential file* is created by sorting the tuples of a relation by their primary key. The tuples are then distributed among blocks, in this order.

Example 1: Fig 2 shows a sequential file on the right. We imagine that keys are integers; we show only the key field, and we make the atypical assumption that there is room for only two records in one block. For instance, the first block of the file holds the records with keys 10 and 20. In this and several other examples, we use integers that are sequential multiples of 10 as keys, although there is surely no requirement that keys form an arithmetic sequence. □

Although in Example 1 we supposed that records were packed as tightly as possible into blocks, it is common to leave some space initially in each block to accomodate new tuples that may be added to a relation. Alternatively, we may accomodate new tuples with overflow blocks.

1.2 Dense Indexes

If records are sorted, we can build on them a *dense index*, which is a sequence of blocks holding only the keys of the records and pointers to the records themselves. The index blocks of the dense index maintain these keys in the same sorted order as in the file itself. Since keys and pointers presumably take much less space than complete records, we expect to use many fewer blocks for the index than for the file itself. The index is especially advantageous when it, but not the data file, can fit in main memory. Then, by using the index, we can find any record given its search key, with only one disk I/O per lookup.

Example 2: Figure 2 suggests a dense index on a sorted file. The first index block contains pointers to the first four records (an atypically small number of pointers for one block), the second block has pointers to the next four, and so on. □

The dense index supports queries that ask for records with a given search-key value. Given key value K, we search the index blocks for K, and when we find it, we follow the associated pointer to the record with key K. It might appear that we need to examine every block of the index, or half the blocks of the index, on average, before we find K. However, there are several factors that make the index-based search more efficient than it seems.

1. The number of index blocks is usually small compared with the number of data blocks.

2. Since keys are sorted, we can use binary search to find K. If there are n blocks of the index, we only look at $\log_2 n$ of them.

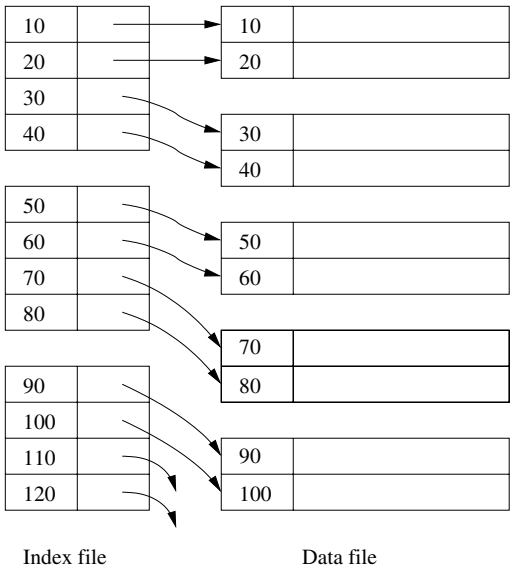

Figure 2: A dense index (left) on a sequential data file (right)

3. The index may be small enough to be kept permanently in main memory buffers. If so, the search for key K involves only main-memory accesses, and there are no expensive disk I/O's to be performed.

1.3 Sparse Indexes

A sparse index typically has only one key-pointer pair per block of the data file. It thus uses less space than a dense index, at the expense of somewhat more time to find a record given its key. You can only use a sparse index if the data file is sorted by the search key, while a dense index can be used for any search key. Figure 3 shows a sparse index with one key-pointer per data block. The keys are for the first records on each data block.

Example 3: As in Example 2, we assume that the data file is sorted, and keys are all the integers divisible by 10, up to some large number. We also continue to assume that four key-pointer pairs fit on an index block. Thus, the first sparse-index block has entries for the first keys on the first four blocks, which are 10, 30, 50, and 70. Continuing the assumed pattern of keys, the second index block has the first keys of the fifth through eighth blocks, which we assume are 90, 110, 130, and 150. We also show a third index block with first keys from the hypothetical ninth through twelfth data blocks. □

To find the record with search-key value K, we search the sparse index for the largest key less than or equal to K. Since the index file is sorted by key, a

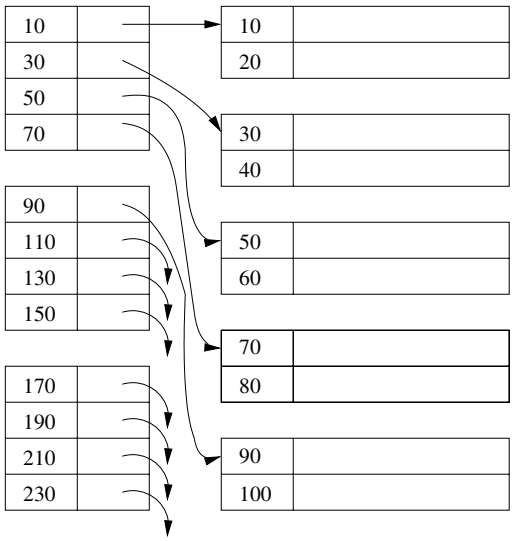

Figure 3: A sparse index on a sequential file

binary search can locate this entry. We follow the associated pointer to a data block. Now, we must search this block for the record with key K. Of course the block must have enough format information that the records and their contents can be identified.

1.4 Multiple Levels of Index

An index file can cover many blocks. Even if we use binary search to find the desired index entry, we still may need to do many disk I/O's to get to the record we want. By putting an index on the index, we can make the use of the first level of index more efficient.

Figure 4 extends Fig. 3 by adding a second index level (as before, we assume keys are every multiple of 10). The same idea would let us place a third-level index on the second level, and so on. However, this idea has its limits, and we prefer the B-tree structure described in Section 2 over building many levels of index.

In this example, the first-level index is sparse, although we could have chosen a dense index for the first level. However, the second and higher levels must be sparse. The reason is that a dense index on an index would have exactly as many key-pointer pairs as the first-level index, and therefore would take exactly as much space as the first-level index.

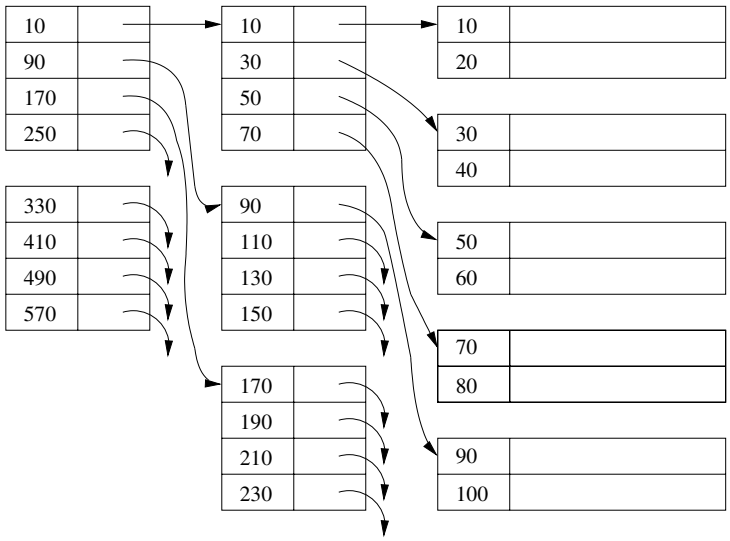

Figure 4: Adding a second level of sparse index

1.5 Secondary Indexes

A secondary index serves the purpose of any index: it is a data structure that facilitates finding records given a value for one or more fields. However, the secondary index is distinguished from the primary index in that a secondary index does not determine the placement of records in the data file. Rather, the secondary index tells us the current locations of records; that location may have been decided by a primary index on some other field. An important consequence of the distinction between primary and secondary indexes is that:

- Secondary indexes are always dense. It makes no sense to talk of a sparse, secondary index. Since the secondary index does not influence location, we could not use it to predict the location of any record whose key was not mentioned in the index file explicitly.

Example 4: Figure 5 shows a typical secondary index. The data file is shown with two records per block, as has been our standard for illustration. The records have only their search key shown; this attribute is integer valued, and as before we have taken the values to be multiples of 10. Notice that, unlike the data file in Fig. 2, here the data is not sorted by the search key.

However, the keys in the index file *are* sorted. The result is that the pointers in one index block can go to many different data blocks, instead of one or a few consecutive blocks. For example, to retrieve all the records with search key 20, we not only have to look at two index blocks, but we are sent by their pointers to three different data blocks. Thus, using a secondary index may result in

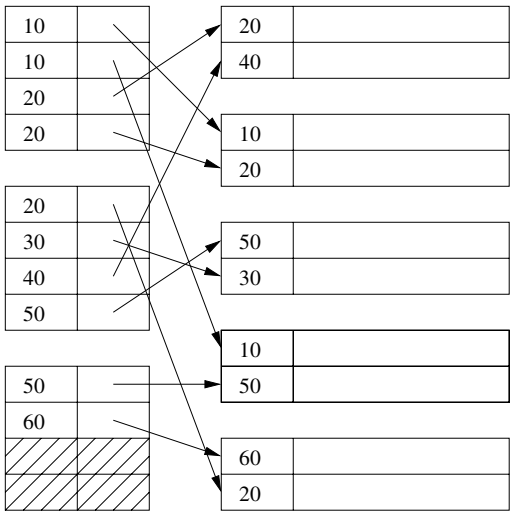

Figure 5: A secondary index

many more disk I/O's than if we get the same number of records via a primary index. However, there is no help for this problem; we cannot control the order of tuples in the data block, because they are presumably ordered according to some other attribute(s). □

1.6 Applications of Secondary Indexes

Besides supporting additional indexes on relations that are organized as sequential files, there are some data structures where secondary indexes are needed for even the primary key. One of these is the "heap" structure, where the records of the relation are kept in no particular order.

A second common structure needing secondary indexes is the *clustered file*. Suppose there are relations R and S, with a many-one relationship from the tuples of R to tuples of S. It may make sense to store each tuple of R with the tuple of S to which it is related, rather than according to the primary key of R. An example will illustrate why this organization makes good sense in special situations.

Example 5: Consider our standard movie and studio relations:

```
Movie(title, year, length, genre, studioName, producerC#)
Studio(name, address, presC#)
```

Suppose further that the most common form of query is:

```
SELECT title, year
FROM Movie, Studio
WHERE presC# = zzz AND Movie.studioName = Studio.name;
```

Here, *zzz* represents any possible certificate number for a studio president. That is, given the president of a studio, we need to find all the movies made by that studio.

If we are convinced that the above query is typical, then instead of ordering `Movie` tuples by the primary key `title` and `year`, we can create a *clustered file structure* for both relations `Studio` and `Movie`, as suggested by Fig. 6. Following each `Studio` tuple are all the `Movie` tuples for all the movies owned by that studio.

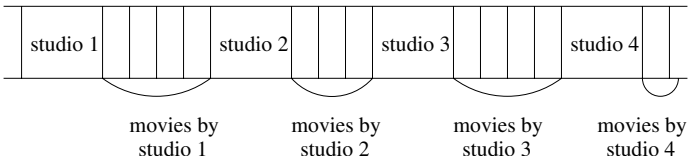

Figure 6: A clustered file with each studio clustered with the movies made by that studio

If we create an index for `Studio` with search key `presC#`, then whatever the value of *zzz* is, we can quickly find the tuple for the proper studio. Moreover, all the `Movie` tuples whose value of attribute `studioName` matches the value of `name` for that studio will follow the studio's tuple in the clustered file. As a result, we can find the movies for this studio by making almost as few disk I/O's as possible. The reason is that the desired `Movie` tuples are packed almost as densely as possible onto the following blocks. However, an index on any attribute(s) of `Movie` would have to be a secondary index. □

1.7 Indirection in Secondary Indexes

There is some wasted space, perhaps a significant amount of wastage, in the structure suggested by Fig. 5. If a search-key value appears n times in the data file, then the value is written n times in the index file. It would be better if we could write the key value once for all the pointers to data records with that value.

A convenient way to avoid repeating values is to use a level of indirection, called *buckets*, between the secondary index file and the data file. As shown in Fig. 7, there is one pair for each search key K. The pointer of this pair goes to a position in a "bucket file," which holds the "bucket" for K. Following this position, until the next position pointed to by the index, are pointers to all the records with search-key value K.

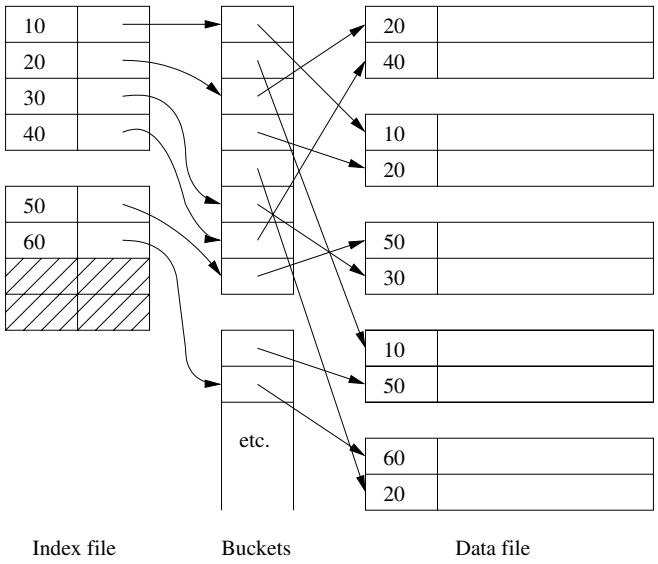

Figure 7: Saving space by using indirection in a secondary index

Example 6: For instance, let us follow the pointer from search key 50 in the index file of Fig. 7 to the intermediate "bucket" file. This pointer happens to take us to the last pointer of one block of the bucket file. We search forward, to the first pointer of the next block. We stop at that point, because the next pointer of the index file, associated with search key 60, points to the next record in the bucket file. □

The scheme of Fig. 7 saves space as long as search-key values are larger than pointers, and the average key appears at least twice. However, even if not, there is an important advantage to using indirection with secondary indexes: often, we can use the pointers in the buckets to help answer queries without ever looking at most of the records in the data file. Specifically, when there are several conditions to a query, and each condition has a secondary index to help it, we can find the bucket pointers that satisfy all the conditions by intersecting sets of pointers in memory, and retrieving only the records pointed to by the surviving pointers. We thus save the I/O cost of retrieving records that satisfy some, but not all, of the conditions.[1]

Example 7: Consider the usual `Movie` relation:

```
Movie(title, year, length, genre, studioName, producerC#)
```

[1]We also could use this pointer-intersection trick if we got the pointers directly from the index, rather than from buckets.

Suppose we have secondary indexes with indirect buckets on both `studioName` and `year`, and we are asked the query

```
SELECT title
FROM Movie
WHERE studioName = 'Disney' AND year = 2005;
```

that is, find all the Disney movies made in 2005.

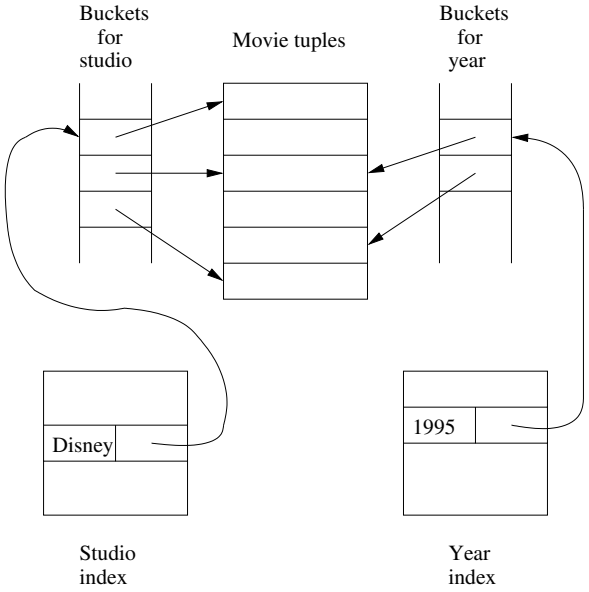

Figure 8: Intersecting buckets in main memory

Figure 8 shows how we can answer this query using the indexes. Using the index on `studioName`, we find the pointers to all records for Disney movies, but we do not yet bring any of those records from disk to memory. Instead, using the index on `year`, we find the pointers to all the movies of 2005. We then intersect the two sets of pointers, getting exactly the movies that were made by Disney in 2005. Finally, we retrieve from disk all data blocks holding one or more of these movies, thus retrieving the minimum possible number of blocks. □

1.8 Document Retrieval and Inverted Indexes

For many years, the information-retrieval community has dealt with the storage of documents and the efficient retrieval of documents with a given set of keywords. With the advent of the World-Wide Web and the feasibility of keeping