8.3 Indexes in SQL

An *index* on an attribute A of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A. We could think of the index as a binary search tree of (key, value) pairs, in which a key a (one of the values that attribute A may have) is associated with a "value" that is the set of locations of the tuples that have a in the component for attribute A. Such an index may help with queries in which the attribute A is compared with a constant, for instance $A = 3$, or even $A \leq 3$. Note that the key for the index can be any attribute or set of attributes, and need not be the key for the relation on which the index is built. We shall refer to the attributes of the index as the *index key* when a distinction needs to be made.

The technology of implementing indexes on large relations is of central importance in the implementation of DBMS's. The most important data structure used by a typical DBMS is the "B-tree," which is a generalization of a balanced binary tree. We shall take up B-trees when we talk about DBMS implementation, but for the moment, thinking of indexes as binary search trees will suffice.

8.3.1 Motivation for Indexes

When relations are very large, it becomes expensive to scan all the tuples of a relation to find those (perhaps very few) tuples that match a given condition. For example, consider the first query we examined:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

from Example 6.1. There might be 10,000 `Movies` tuples, of which only 200 were made in 1990.

The naive way to implement this query is to get all 10,000 tuples and test the condition of the `WHERE` clause on each. It would be much more efficient if we had some way of getting only the 200 tuples from the year 1990 and testing each of them to see if the studio was Disney. It would be even more efficient if we could obtain directly only the 10 or so tuples that satisfied both the conditions of the `WHERE` clause — that the studio is Disney and the year is 1990; see the discussion of "multiattribute indexes," in Section 8.3.2.

Indexes may also be useful in queries that involve a join. The following example illustrates the point.

Example 8.9: Recall the query

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

from Example 6.12 that asks for the name of the producer of *Star Wars*. If there is an index on `title` of `Movies`, then we can use this index to get the tuple for *Star Wars*. From this tuple, we can extract the `producerC#` to get the certificate of the producer.

Now, suppose that there is also an index on `cert#` of `MovieExec`. Then we can use the `producerC#` with this index to find the tuple of `MovieExec` for the producer of *Star Wars*. From this tuple, we can extract the producer's name. Notice that with these two indexes, we look at only the two tuples, one from each relation, that are needed to answer the query. Without indexes, we have to look at every tuple of the two relations. □

8.3.2 Declaring Indexes

Although the creation of indexes is not part of any SQL standard up to and including SQL-99, most commercial systems have a way for the database designer to say that the system should create an index on a certain attribute for a certain relation. The following syntax is typical. Suppose we want to have an index on attribute `year` for the relation `Movies`. Then we say:

```
CREATE INDEX YearIndex ON Movies(year);
```

The result will be that an index whose name is `YearIndex` will be created on attribute `year` of the relation `Movies`. Henceforth, SQL queries that specify a year may be executed by the SQL query processor in such a way that only those tuples of `Movies` with the specified year are ever examined; there is a resulting decrease in the time needed to answer the query.

Often, a DBMS allows us to build a single index on multiple attributes. This type of index takes values for several attributes and efficiently finds the tuples with the given values for these attributes.

Example 8.10: Since `title` and `year` form a key for `Movies`, we might expect it to be common that values for both these attributes will be specified, or neither will. The following is a typical declaration of an index on these two attributes:

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

Since (`title`, `year`) is a key, if follows that when we are given a title and year, we know the index will find only one tuple, and that will be the desired tuple. In contrast, if the query specifies both the title and year, but only `YearIndex` is available, then the best the system can do is retrieve all the movies of that year and check through them for the given title.

If, as is often the case, the key for the multiattribute index is really the concatenation of the attributes in some order, then we can even use this index to find all the tuples with a given value in the first of the attributes. Thus, part of the design of a multiattribute index is the choice of the order in which the attributes are listed. For instance, if we were more likely to specify a title than a year for a movie, then we would prefer to order the attributes as above; if a year were more likely to be specified, then we would ask for an index on (year, title). □

If we wish to delete the index, we simply use its name in a statement like:

```
DROP INDEX YearIndex;
```

8.3.3 Exercises for Section 8.3

Exercise 8.3.1: For our running movies example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Declare indexes on the following attributes or combination of attributes:

a) studioName.

b) address of MovieExec.

c) genre and length.

8.4 Selection of Indexes

Choosing which indexes to create requires the database designer to analyze a trade-off. In practice, this choice is one of the principal factors that influence whether a database design gives acceptable performance. Two important factors to consider are:

- The existence of an index on an attribute may speed up greatly the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well.

- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

8.4.1 A Simple Cost Model

To understand how to choose indexes for a database, we first need to know where the time is spent answering a query. The details of how relations are stored will be taken up when we consider DBMS implementation. But for the moment, let us state that the tuples of a relation are normally distributed among many pages of a disk.[1] One page, which is typically several thousand bytes at least, will hold many tuples.

To examine even one tuple requires that the whole page be brought into main memory. On the other hand, it costs little more time to examine all the tuples on a page than to examine only one. There is a great time saving if the page you want is already in main memory, but for simplicity we shall assume that never to be the case, and every page we need must be retrieved from the disk.

8.4.2 Some Useful Indexes

Often, the most useful index we can put on a relation is an index on its key. There are two reasons:

1. Queries in which a value for the key is specified are common. Thus, an index on the key will get used frequently.

2. Since there is at most one tuple with a given key value, the index returns either nothing or one location for a tuple. Thus, at most one page must be retrieved to get that tuple into main memory (although there may be other pages that need to be retrieved to use the index itself).

The following example shows the power of key indexes, even in a query that involves a join.

Example 8.11: Recall Figure 6.3, where we suggested an exhaustive pairing of tuples of `Movies` and `MovieExec` to compute a join. Implementing the join this way requires us to read each of the pages holding tuples of `Movies` and each of the pages holding tuples of `MovieExec` at least once. In fact, since these pages may be too numerous to fit in main memory at the same time, we may have to read each page from disk many times. With the right indexes, the whole query might be done with as few as two page reads.

An index on the key `title` and `year` for `Movies` would help us find the one `Movies` tuple for *Star Wars* quickly. Only one page — the page containing that tuple — would be read from disk. Then, after finding the producer-certificate number in that tuple, an index on the key `cert#` for `MovieExec` would help us quickly find the one tuple for the producer in the `MovieExec` relation. Again, only one page with `MovieExec` tuples would be read from disk, although we might need to read a small number of other pages to use the `cert#` index. □

When the index is not on a key, it may or may not be able to improve the time spent retrieving from disk the tuples needed to answer a query. There are two situations in which an index can be effective, even if it is not on a key.

[1] Pages are usually referred to as "blocks" in discussion of databases, but if you are familiar with a paged-memory system from operating systems you should think of the disk as divided into pages.

1. If the attribute is almost a key; that is, relatively few tuples have a given value for that attribute. Even if each of the tuples with a given value is on a different page, we shall not have to retrieve many pages from disk.

2. If the tuples are "clustered" on that attribute. We *cluster* a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible. Then, even if there are many tuples, we shall not have to retrieve nearly as many pages as there are tuples.

Example 8.12: As an example of an index of the first kind, suppose `Movies` had an index on `title` rather than `title` and `year`. Since `title` by itself is not a key for the relation, there would be titles such as *King Kong*, where several tuples matched the index key `title`. If we compared use of the index on `title` with what happens in Example 8.11, we would find that a search for movies with title *King Kong* would produce three tuples (because there are three movies with that title, from years 1933, 1976, and 2005). It is possible that these tuples are on three different pages, so all three pages would be brought into main memory, roughly tripling the amount of time this step takes. However, since the relation `Movies` probably is spread over many more than three pages, there is still a considerable time saving in using the index.

At the next step, we need to get the three `producerC#` values from these three tuples, and find in the relation `MovieExec` the producers of these three movies. We can use the index on `cert#` to find the three relevant tuples of `MovieExec`. Possibly they are on three different pages, but we still spend less time than we would if we had to bring the entire `MovieExec` relation into main memory. □

Example 8.13: Now, suppose the only index we have on `Movies` is one on `year`, and we want to answer the query:

```
SELECT *
FROM Movies
WHERE year = 1990;
```

First, suppose the tuples of `Movies` are not clustered by year; say they are stored alphabetically by title. Then this query gains little from the index on `year`. If there are, say, 100 movies per page, there is a good chance that any given page has at least one movie made in 1990. Thus, a large fraction of the pages used to hold the relation `Movies` will have to be brought to main memory.

However, suppose the tuples of `Movies` are clustered on `year`. Then we could use the index on `year` to find only the small number of pages that contained tuples with `year` = 1990. In this case, the `year` index will be of great help. In comparison, an index on the combination of `title` and `year` would be of little help, no matter what attribute or attributes we used to cluster `Movies`. □

8.4.3 Calculating the Best Indexes to Create

It might seem that the more indexes we create, the more likely it is that an index useful for a given query will be available. However, if modifications are the most frequent action, then we should be very conservative about creating indexes. Each modification on a relation R forces us to change any index on one or more of the modified attributes of R. Thus, we must read and write not only the pages of R that are modified, but also read and write certain pages that hold the index. But even when modifications are the dominant form of database action, it may be an efficiency gain to create an index on a frequently used attribute. In fact, since some modification commands involve querying the database (e.g., an `INSERT` with a select-from-where subquery or a `DELETE` with a condition) one must be very careful how one estimates the relative frequency of modifications and queries.

Remember that the typical relation is stored over many disk blocks (pages), and the principal cost of a query or modification is often the number of pages that need to be brought to main memory. Thus, indexes that let us find a tuple without examining the entire relation can save a lot of time. However, the indexes themselves have to be stored, at least partially, on disk, so accessing and modifying the indexes themselves cost disk accesses. In fact, modification, since it requires one disk access to read a page and another disk access to write the changed page, is about twice as expensive as accessing the index or the data in a query.

To calculate the new value of an index, we need to make assumptions about which queries and modifications are most likely to be performed on the database. Sometimes, we have a history of queries that we can use to get good information, on the assumption that the future will be like the past. In other cases, we may know that the database supports a particular application or applications, and we can see in the code for those applications all the SQL queries and modifications that they will ever do. In either situation, we are able to list what we expect are the most common query and modification forms. These forms can have variables in place of constants, but should otherwise look like real SQL statements. Here is a simple example of the process, and of the calculations that we need to make.

Example 8.14: Let us consider the relation

```
StarsIn(movieTitle, movieYear, starName)
```

Suppose that there are three database operations that we sometimes perform on this relation:

Q_1: We look for the title and year of movies in which a given star appeared. That is, we execute a query of the form:

```
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = s;
```

for some constant s.

Q_2: We look for the stars that appeared in a given movie. That is, we execute a query of the form:

```
SELECT starName
FROM StarsIn
WHERE movieTitle = t AND movieYear = y;
```

for constants t and y.

I: We insert a new tuple into `StarsIn`. That is, we execute an insertion of the form:

```
INSERT INTO StarsIn VALUES(t, y, s);
```

for constants t, y, and s.

Let us make the following assumptions about the data:

1. `StarsIn` occupies 10 pages, so if we need to examine the entire relation the cost is 10.

2. On the average, a star has appeared in 3 movies and a movie has 3 stars.

3. Since the tuples for a given star or a given movie are likely to be spread over the 10 pages of `StarsIn`, even if we have an index on `starName` or on the combination of `movieTitle` and `movieYear`, it will take 3 disk accesses to find the (average of) 3 tuples for a star or movie. If we have no index on the star or movie, respectively, then 10 disk accesses are required.

4. One disk access is needed to read a page of the index every time we use that index to locate tuples with a given value for the indexed attribute(s). If an index page must be modified (in the case of an insertion), then another disk access is needed to write back the modified page.

5. Likewise, in the case of an insertion, one disk access is needed to read a page on which the new tuple will be placed, and another disk access is needed to write back this page. We assume that, even without an index, we can find some page on which an additional tuple will fit, without scanning the entire relation.

Figure 8.3 gives the costs of each of the three operations; Q_1 (query given a star), Q_2 (query given a movie), and I (insertion). If there is no index, then we must scan the entire relation for Q_1 or Q_2 (cost 10),[2] while an insertion requires

[2]There is a subtle point that we shall ignore here. In many situations, it is possible to store a relation on disk using consecutive pages or tracks. In that case, the cost of retrieving the entire relation may be significantly less than retrieving the same number of pages chosen randomly.

Action	No Index	Star Index	Movie Index	Both Indexes
Q_1	10	4	10	4
Q_2	10	10	4	4
I	2	4	4	6
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

Figure 8.3: Costs associated with the three actions, as a function of which indexes are selected

merely that we access a page with free space and rewrite it with the new tuple (cost of 2, since we assume that page can be found without an index). These observations explain the column labeled "No Index."

If there is an index on stars only, then Q_2 still requires a scan of the entire relation (cost 10). However, Q_1 can be answered by accessing one index page to find the three tuples for a given star and then making three more accesses to find those tuples. Insertion I requires that we read and write both a page for the index and a page for the data, for a total of 4 disk accesses.

The case where there is an index on movies only is symmetric to the case for stars only. Finally, if there are indexes on both stars and movies, then it takes 4 disk accesses to answer either Q_1 or Q_2. However, insertion I requires that we read and write two index pages as well as a data page, for a total of 6 disk accesses. That observation explains the last column in Fig. 8.3.

The final row in Fig. 8.3 gives the average cost of an action, on the assumption that the fraction of the time we do Q_1 is p_1 and the fraction of the time we do Q_2 is p_2; therefore, the fraction of the time we do I is $1 - p_1 - p_2$.

Depending on p_1 and p_2, any of the four choices of index/no index can yield the best average cost for the three actions. For example, if $p_1 = p_2 = 0.1$, then the expression $2+8p_1+8p_2$ is the smallest, so we would prefer not to create any indexes. That is, if we are doing mostly insertion, and very few queries, then we don't want an index. On the other hand, if $p_1 = p_2 = 0.4$, then the formula $6 - 2p_1 - 2p_2$ turns out to be the smallest, so we would prefer indexes on both `starName` and on the (`movieTitle, movieYear`) combination. Intuitively, if we are doing a lot of queries, and the number of queries specifying movies and stars are roughly equally frequent, then both indexes are desired.

If we have $p_1 = 0.5$ and $p_2 = 0.1$, then an index on stars only gives the best average value, because $4 + 6p_2$ is the formula with the smallest value. Likewise, $p_1 = 0.1$ and $p_2 = 0.5$ tells us to create an index on only movies. The intuition is that if only one type of query is frequent, create only the index that helps that type of query. □

8.4.4 Automatic Selection of Indexes to Create

"Tuning" a database is a process that includes not only index selection, but the choice of many different parameters. We have not yet discussed much about physical implementation of databases, but some examples of tuning issues are the amount of main memory to allocate to various processes and the rate at which backups and checkpoints are made (to facilitate recovery from a crash). There are a number of tools that have been designed to take the responsibility from the database designer and have the system tune itself, or at least advise the designer on good choices.

We shall mention some of these projects in the bibliographic notes for this chapter. However, here is an outline of how the index-selection portion of tuning advisors work.

1. The first step is to establish the query workload. Since a DBMS normally logs all operations anyway, we may be able to examine the log and find a set of representative queries and database modifications for the database at hand. Or it is possible that we know, from the application programs that use the database, what the typical queries will be.

2. The designer may be offered the opportunity to specify some constraints, e.g., indexes that must, or must not, be chosen.

3. The tuning advisor generates a set of possible *candidate* indexes, and evaluates each one. Typical queries are given to the query optimizer of the DBMS. The query optimizer has the ability to estimate the running times of these queries under the assumption that one particular set of indexes is available.

4. The index set resulting in the lowest cost for the given workload is suggested to the designer, or it is automatically created.

A subtle issue arises when we consider possible indexes in step (3). The existence of previously chosen indexes may influence how much *benefit* (improvement in average execution time of the query mix) another index offers. A "greedy" approach to choosing indexes has proven effective.

a) Initially, with no indexes selected, evaluate the benefit of each of the candidate indexes. If at least one provides positive benefit (i.e., it reduces the average execution time of queries), then choose that index.

b) Then, reevaluate the benefit of each of the remaining candidate indexes, assuming that the previously selected index is also available. Again, choose the index that provides the greatest benefit, assuming that benefit is positive.

c) In general, repeat the evaluation of candidate indexes under the assumption that all previously selected indexes are available. Pick the index with maximum benefit, until no more positive benefits can be obtained.

8.4.5 Exercises for Section 8.4

Exercise 8.4.1: Suppose that the relation `StarsIn` discussed in Example 8.14 required 100 pages rather than 10, but all other assumptions of that example continued to hold. Give formulas in terms of p_1 and p_2 to measure the cost of queries Q_1 and Q_2 and insertion I, under the four combinations of index/no index discussed there.

! **Exercise 8.4.2:** In this problem, we consider indexes for the relation

`Ships(name, class, launched)`

from our running battleships exercise. Assume:

i. `name` is the key.

ii. The relation `Ships` is stored over 50 pages.

iii. The relation is clustered on `class` so we expect that only one disk access is needed to find the ships of a given class.

iv. On average, there are 5 ships of a class, and 25 ships launched in any given year.

v. With probability p_1 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE name =` n.

vi. With probability p_2 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE class =` c.

vii. With probability p_3 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE launched =` y.

viii. With probability $1 - p_1 - p_2 - p_3$ the operation on this relation is an insertion of a new tuple into `Ships`.

You can also make the assumptions about accessing indexes and finding empty space for insertions that were made in Example 8.14.

Consider the creation of indexes on `name`, `class`, and `launched`. For each combination of indexes, estimate the average cost of an operation. As a function of p_1, p_2, and p_3, what is the best choice of indexes?

8.5 Materialized Views

A view describes how a new relation can be constructed from base tables by executing a query on those tables. Until now, we have thought of views only as logical descriptions of relations. However, if a view is used frequently enough, it may even be efficient to *materialize* it; that is, to maintain its value at all times. As with maintaining indexes, there is a cost involved in maintaining a materialized view, since we must recompute parts of the materialized view each time one of the underlying base tables changes.