Topic 5.2: How Indexes Work

An index is a data structure that improves the speed of locating records that are to be retrieved, updated, or deleted. Indexes can also improve the performance of inserts when primary key or unique constraints have been defined because they support a fast and efficient way of determining if the new record would be a duplicate. While indexes are used for all sorts of file and database systems, I will use relational database indexes here as an example.

Without indexes, the DBMS must read through all the rows in order to find the rows required by an SQL statement (a process known as a *full table scan*). This technique works well if the tables are relatively small (a few thousand rows or less), or for queries that are going to require many of the rows in a table. However, indexes are essential in order to efficiently find a single row or a relatively small set of rows in larger tables.

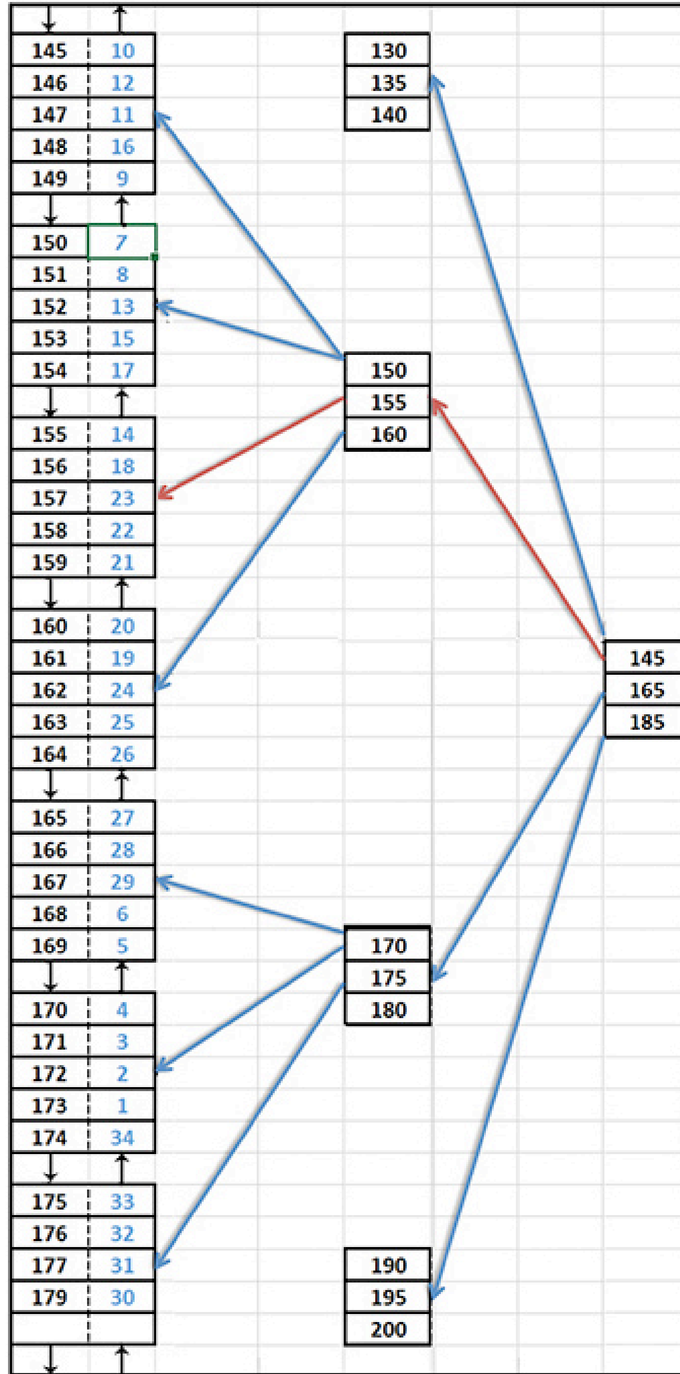
While indexes offer performance advantages, they also come with a price in that they require storage and must be maintained. Storage costs seem to become less of a concern with each passing day because storage devices continue to decline in price. However, index maintenance should not be overlooked because it slows the performance of inserts, updates, and deletes. When new rows are inserted, index records must be created for any columns that are indexed, with the exception of null values, which are never indexed. Similarly, when indexed columns are updated, the indexes must be updated as well. And when rows are deleted, any index records for the row must also be deleted.

B-tree Indexes

B-tree (balanced tree) indexes are the most commonly used indexes in relational databases. As the name suggests, b-tree indexes have an inverted tree structure with a *root* node at the top of the structure, *branch layer* nodes in the middle of the structure, and the *leaf layer* nodes at the bottom of the structure. Nodes are connected using relative block address (RBA) pointers. (Recall that an RBA pointer consists of the value of the relative block within a file that contains the desired record. For example, an RBA address of 3857 would tell the DBMS to look in block 3,857 of the file for the desired record.)

The leaf layer node contains one logical record for each indexed value, which is maintained in key sequence to allow the DBMS to process table rows in key sequence by scanning the leaf layer. Each record in the leaf layer consists of the indexed value, along with the RowID of the table row that contains the value. A *RowID* is the address of a row within a database file, which the DBMS uses to quickly and efficiently locate a row within a table. In most RDBMS implementations, RowID values are invisible to database users, but some RDBMS products make them available in SQL queries using a pseudo column. (In SQL, a *pseudo column* behaves like a column, but is not actually stored within the table. CURRENT_DATE is a common example of a pseudo column.) You can think of the leaf layer of an index as being like the index in the back of a book, where keywords (indexed values) are listed in alphabetical sequence, along with a page number where the term appears in the book (similar to the RowID pointing to a table row where the value appears).

As an example, let's look at a table named EMPLOYEE. The primary key is EMPLOYEE_ID, with values ranging from 130 through 200. The following diagram shows the b-tree index for EMPLOYEE_ID, followed by a partial listing of the rows in the EMPLOYEE table.



B-tree Index

ROWID	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT
1	**173**	Sundita	Kumar	6100	0.1
2	**172**	Elizabeth	Bates	7300	0.15

3	**171**	William	Smith	7400	0.15
4	**170**	Tayler	Fox	9600	0.2
5	**169**	Harrison	Bloom	10000	0.2
6	**168**	Lisa	Ozer	11500	0.25
7	**150**	Peter	Tucker	10000	0.3
8	**151**	David	Bernstein	9500	0.25
9	**149**	Eleni	Zlotkey	10500	0.2
10	**145**	John	Russell	14000	0.4
11	**147**	Alberto	Errazuriz	12000	0.3
12	**146**	Karen	Partners	13500	0.3
13	**152**	Peter	Hall	9000	0.25
14	**155**	Oliver	Tuvault	7000	0.15
15	**153**	Christopher	Olsen	8000	0.2
16	**148**	Gerald	Cambrault	11000	0.3
17	**154**	Nanette	Cambrault	7500	0.2
18	**156**	Janette	King	10000	0.35
19	**161**	Sarath	Sewall	7000	0.25
20	**160**	Louise	Doran	7500	0.3
21	**159**	Lindsey	Smith	8000	0.3
22	**158**	Allan	McEwen	9000	0.35
23	**157**	Patrick	Sully	9500	0.35
24	**162**	Clara	Vishney	10500	0.25
25	**163**	Danielle	Greene	9500	0.15
26	**164**	Mattea	Marvins	7200	0.1
27	**165**	David	Lee	6800	0.1
28	**166**	Sundar	Ande	6400	0.1
29	**167**	Amit	Banda	6200	0.1
30	**179**	Charles	Johnson	6200	0.1
31	**177**	Jack	Livingston	8400	0.2
32	**176**	Jonathon	Taylor	8600	0.2
33	**175**	Alyssa	Hutton	8800	0.25
34	**174**	Ellen	Abel	11000	0.3

The top level of the index, known as the root, contains 3 values: 145, 165, and 185. From this top level there are four RBA pointers to blocks in the next level of the index, which is a *branch* layer. These pointers represent the path that the DBMS will follow when searching for values that are, respectively, less than 145; greater than or equal to 145, but less than 165; greater than or equal to 165, but less than 185; and greater than 185.

Similarly, each node in the branch layer contains 3 values, with 4 pointers going to the next layer. In this example, there is only one branch layer, so the next layer is the leaf layer. However, if more rows were added to the EMPLOYEE table, we would exceed the currently supported range of EMPLOYEE_ID values (130 through 200), which would require the DBMS to split the branch layer into two layers. Index maintenance such as this occurs automatically because the DBMS takes full responsibility for maintaining indexes. (On the diagram, pointers that would go from the branch layer to the leaf layer nodes that are off the page are not

shown. If all of the leaf layer nodes were shown, then each branch layer node would have four pointers going to the leaf layer.) The term "balanced tree" comes from the fact that all the nodes at the same layer have the same number elements and pointers; hence they "balance" with one another.

As mentioned, each node in the leaf layer has logical records that contain key values (EMPLOYEE_ID values in this example) and the RowID of the corresponding row in the table. In this case, the index is a primary key index, and therefore unique, so each EMPLOYEE_ID value appears only once. For a non-unique index, there are two possible ways to handle multiple RowID values for each key value. One method is to store a list of RowID values with each key value, and the other is to repeat the key value in an additional leaf layer record such that each record still contains a single value for the key along with a single value for the RowID. Note also that leaf nodes are "chained" together using RBA pointers to the next leaf node as well as the previous leaf node. These pointers allow leaf nodes to be scanned sequentially by the DBMS.

To help you understand how an index is searched, let's follow the path for this query:

```
SELECT LAST_NAME
  FROM EMPLOYEE
 WHERE EMPLOYEE_ID = 157;
```

The query optimizer in the DBMS sees that there is an index on the EMPLOYEE_ID column and thus uses the index instead of scanning the table. The SQL engine within the DBMS looks at the root level of the index, and since the value being sought is greater than or equal to 145, but less than 165, it follows the second pointer (the one shown in red on the diagram) to the branch layer node that contains the values 150, 155, and 160. Inspecting those values, it follows the third pointer (again shown in red) because 157 is greater than or equal to 155, but less than 160. On reaching the leaf node, it searches the block for the desired record, which is the third one in the block. This record contains key value 157 and RowID 23 (shown in blue). Now the DBMS knows a matching table row exists, and thus accesses the table to retrieve row 23 and return the LAST_NAME value of Sully.

Compare this to the prospect of scanning rows to find a randomly selected key value. The table has rows for employees 130 through 200, which means there can be up to 71 rows. In this case, assume one row has been deleted, leaving an even 70 rows. If the DBMS scans through from the beginning of the table looking for the row for Employee ID 157, it will, on average, have to read through half the number of rows (35 in this case), to find it. Yet, with the index, it is able to find the row in just 3 reads (the root node, the branch node, and the leaf node), following by scanning just 3 records in the leaf node, and finally 1 more read to retrieve the table row. At larger scale, such as a table with several hundred thousand rows, the performance improvement using an index is even more profound.

Let's now look at a somewhat different example where the query looks for a sequential range of key values:

```
SELECT LAST_NAME
  FROM EMPLOYEE
 WHERE EMPLOYEE_ID BETWEEN 157 AND 160;
```

As with the previous example, the SQL engine works its way through the index to find the leaf node record for EMPLOYEE_ID value 157 and accesses the table to find the LAST_NAME value of Sully. However, for the next value after 157, it does not have to go back through the upper layers of the index because the leaf layer is kept in ascending key sequence. All it has to do is process leaf node records in sequence until it finds a value that is greater than 160. (Actually, in this case, it can stop when if it finds the value 160 because the index is unique.) So, in the same leaf node block as the record for EMPLOYEE_ID 157, it finds the records for

EMPLOYEE_IDs 158 and 159, using RowIDs 22 and 21 to find the LAST_NAME values of McEwan and Smith. Then it follows the pointer from the current leaf node to the next one in sequence to find the record for EMPLOYEE_ID 160, using RowID 20 to find the LAST_NAME value of Doran. This sort of processing is known as *skip sequential*, where the DBMS uses the index to "skip" into the leaf layer using the first value in a range, and then process sequentially forward or backward to find the remaining values.

Although this example shows 3 values and 4 pointers in each node of the index (a pattern known as a 3-4 index), b-tree indexes can be created using as few as 1 value and 2 pointers from each node (a pattern known as a *binary tree*, which, unfortunately is also known as a b-tree), or an index could use more than 3 values and 4 pointers per node. The choice is purely up to the designers of the DBMS and the underlying file system.

Also, this example shows table rows maintained in random sequence instead of primary key sequence. This is the most typical implementation. However, some vendors offer what are known as *clustered indexes*, which are used to maintain the base table in key sequence. Obviously, there can be only one clustered index per table. The advantage of clustered indexes is they can keep a table in a sequence that optimizes joins. For example, an ORDER_LINE_ITEM table can be kept in ORDER_ID sequence to improve joins with an ORDER table. However, this comes with a performance cost that can be significant because every time a row is inserted into a table with a clustered index, other rows in the table must be moved around to make space for the newly added row (since it must be in ascending key sequence).

As mentioned earlier, the DBMS must automatically maintain the index. This includes removing leaf layer records (or marking them as logically deleted) when rows are deleted from the table. If you look at the rightmost leaf node in the diagram, you will see that the key value 180 and its corresponding RowID value are missing. This represents a deleted record. If a large number of deleted records exist in the leaf layer, index performance can suffer. As rows are inserted into the table, corresponding records must be created in the index. When a node reaches the point where no more records can fit, the DBMS must split it into two and adjust the branch level nodes above it accordingly. Over time an index can become quite fragmented if there are a lot of deletes and inserts, or if the deletes are inserts are concentrated in particular ranges of key values. When framentation occurs, index performance can decline sharply. However, the index can be reorganized (if the DBMS vendor provides an index reorganization utility with the DBMS), or the index can be dropped and recreated, which will return it to its previous level of efficiency.

Effective Use of B-Tree Indexes

B-tree indexes are most effective when they are highly selective. *Index selectivity* is measured as the ratio of key values to the number of rows in the table. A unique index has a selectivity ratio of 1.0 because each row must have a key value and no key values can be repeated. For example, a 100 row table will have 100 key values, and 100/100 = 1.0. An index on a column such as EMPLOYEE_LOCATION would be poor if there were 500 employees and only 2 locations (2/500 = 0.004), but good if there were 350 locations (350/500 = 0.7). Performance rules vary from vendor to vendor, but in general, selectivity of 0.7 and above is considered quite good for b-tree indexes, although some vendors recommend indexes with a selectivity as low as .5. It's best to check the DBMS vendor's performance recommendations.

Assuming they are reasonably selective, b-tree indexes are useful in the following situations:

- Primary key and unique constraints. The DBMS will automatically create indexes to enforce these constraints.

- Foreign key columns when joins are made from the primary key to the foreign key. For example, an index on CUSTOMER_ID in an ORDER table will improve performance of joins that seek all orders for one or more customers.
- Columns frequently used in WHERE clauses of SQL statements. Indexes here will avoid table scans to find matching rows.
- Columns frequently used in ORDER BY or GROUP BY clauses. Indexes here can avoid the need to sort the entire table because the DBMS can use the leaf layer to find all rows in key sequence.
- The larger the table, the less you want any database query to scan all the rows.

However, keep the following in mind:

- As tables are updated, indexes must be maintained. For tables that are used to record everyday business events and transactions, it is best to avoid adding more than 3 or 4 indexes per table.
- For tables smaller than 1,000 rows or so, indexes other than those required to enforce primary key and unique constraints, are seldom useful to the DBMS. Query optimizers estimate the cost of running queries with and without indexes, and if the estimated cost of scanning the table is lower than the cost of going through the index, the DBMS will scan the table instead.
- For tables with short rows, consider an index-organized table (IOT), if supported by the vendor. This option puts all the table column values in the index leaf nodes (instead of the RowID) and thus the entire table is built into the index. Oracle offers this option. IBM's DB2 offers an interesting variation wherein selected table columns can be added to the index, allowing those columns to be returned by the index without the need to access table rows; however, the table still exists for other column values that are less frequently accessed.

Bitmap Indexes

Bitmap indexes are special indexes that use bit arrays (bit maps) to store index values. Recall that a *bit* is a binary digit, which can have only two values – 0 or 1. A *bit array* is merely a sting of bits. All the data in a computerized system is stored as bits, but the bits are typically organized into bytes, each consisting of 8 bits. For character strings, each character takes on byte. For example, in computer systems that use the ASCII character set, the letter A is stored as 01000001, which its lowercase equivalent is stored as 01100001.

With bitmap indexes within an RDBMS, we create a string of bits to represent each possible value of an attribute, with each bit within a string representing a row in the base table. For a table containing 250,000 rows, each bit string would be 250,000 bits long, which sounds large until you realize that 250,000 bits is the equivalent of only 31,250 bytes, or roughly 30.5K.

Bitmap indexes have been used effectively on columns with low cardinality (meaning columns that have relatively few values compared to the number of rows in the table). However, some researchers claim they have been successful using bitmap indexes for medium cardinality columns in very large tables. Current DBMS products that support bitmap indexes include Oracle, DB2, MySQL, Sybase IQ, and HP Vertica.

As an example, let's look at bitmap indexes for a table that contains sales data for a large company. Here is a subset of the data in the table:

RowID	Region	Year/Qtr	Product Line	Transaction ID	Amount
1	South	2015-1	Systems	19574	$12,754
2	North	2015-2	Accessories	84737	$6,274
3	West	2015-4	Software	25141	$47,250
4	East	2015-3	Accessories	85743	$5,280
5	North	2016-1	Systems	64532	$187,205
6	North	2016-1	Parts	32767	$299
7	South	2015-2	Software	12843	$5,000
8	North	2015-1	Parts	48732	$598
9	West	2015-2	Systems	18749	$96,200
10	South	2015-3	Software	79453	$86,327
11	East	2015-1	Systems	63890	$275,900
12	North	2015-4	Accessories	48327	$2,785

The columns that qualify as low cardinality are Region, Year/Qty, and Product Line. Transaction ID is unique for each row, and therefore would likely be a good fit for a b-tree index. The Amount column is also likely to be a very high cardinality based on the sample data, but it could potentially be coded into ranges such as less than 5K, 5K – 10K, 10K – 20K, and so forth, which would be appropriate for a bitmap index. For the purposes of illustrating bitmap indexes, let's look at the Region, Year/Qty, and Product Line columns as straightforward examples.

Here is the bitmap index for a SALES_REGION column, with four possible regions:

REGION	**TABLE ROW**											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
North	0	1	0	0	1	1	0	1	0	0	0	1
South	1	0	0	0	0	0	1	0	0	1	0	0
East	0	0	0	1	0	0	0	0	0	0	1	0
West	0	0	1	0	0	0	0	0	1	0	0	0

Note that each column in the bitmap, which represents one row in the Sales table, has only one bit set to 1. This should make sense when you consider that each row in the Sales table has only one value for the Region column. This characteristic is one of the hallmarks of bitmap indexes.

Here is the bitmap index for the Year/Qtr column:

YEAR/QTR	**TABLE ROW**											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
2015-1	1	0	0	0	0	0	0	1	0	0	1	0
2015-2	0	1	0	0	0	0	1	0	1	0	0	0
2015-3	0	0	0	1	0	0	0	0	0	1	0	0
2015-4	0	0	1	0	0	0	0	0	0	0	0	1
2016-1	0	0	0	0	1	1	0	0	0	0	0	0

Finally, here is the bitmap index for the Product Line column:

PRODUCT LINE	TABLE ROW											
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Accessories	0	1	0	1	0	0	0	0	0	0	0	1
Parts	0	0	0	0	0	1	0	1	0	0	0	0
Software	0	0	1	0	0	0	1	0	0	1	0	0
Systems	1	0	0	0	1	0	0	0	1	0	1	0

The main benefits of bitmap indexes are first, they are much more compact than b-tree indexes when used on low cardinality columns, and second, they can easily be compared with one another using AND and OR conditions. In fact, they are a natural for Boolean algebra because each value in the index contains a logical TRUE (1) or FALSE (0).

On the downside, bitmap indexes are quite expensive to maintain as data values change in the base tables. Therefore, they are most useful in read-only databases, such as business intelligence and reporting databases that contain historical data that is not expected to change. (Even history can change sometimes, such as when a company must restate its sales for a quarter due to an accounting or regulatory issue.) With read-only tables, whenever fresh data is loaded into the tables, the indexes are simply rebuilt. (Business Intelligence Databases are presented in Module 8.)