1 Indexes

1.1 Introduction

“An index makes the query fast” is the most basic explanation of an index I have ever seen. Although it describes the most important aspect of an index very well, it is—unfortunately—not sufficient for this book. This chapter describes the index structure in a less superficial way but doesn't dive too deeply into details. It provides just enough insight for one to understand the SQL performance aspects discussed throughout the book.

An index is a distinct structure in the database that is built using the `create index` statement. It requires its own disk space and holds a copy of the indexed table data. That means that an index is pure redundancy. Creating an index does not change the table data; it just creates a new data structure that refers to the table. A database index is, after all, very much like the index at the end of a book: it occupies its own space, it is highly redundant, and it refers to the actual information stored in a different place.

Searching in a database index is like searching in a printed telephone directory. The key concept is that all entries are arranged in a well-defined order. Finding data in an ordered data set is fast and easy because the sort order determines each entry's position.

A database index is, however, more complex than a printed directory because it undergoes constant change. Updating a printed directory for every change is impossible for the simple reason that there is no space between existing entries to add new ones. A printed directory bypasses this problem by only handling the accumulated updates with the next printing. An SQL database cannot wait that long. It must process `insert`, `delete` and `update` statements immediately, keeping the index order without moving large amounts of data.

The database combines two data structures to meet the challenge: a doubly linked list and a search tree. These two structures explain most of the database's performance characteristics.

1.2 Leaf Nodes

The primary purpose of an index is to provide an ordered representation of the indexed data. It is, however, not possible to store the data sequentially because an `insert` statement would need to move the following entries to make room for the new one. Moving large amounts of data is very time-consuming so the `insert` statement would be very slow. The solution to the problem is to establish a logical order that is independent of physical order in memory.

The logical order is established via a doubly linked list. Every node has links to two neighboring entries, very much like a chain. New nodes are inserted between two existing nodes by updating their links to refer to the new node. The physical location of the new node doesn't matter because the doubly linked list maintains the logical order.

The data structure is called a doubly linked list because each node refers to the preceding and the following node. It enables the database to read the index forwards or backwards as needed. It is thus possible to insert new entries without moving large amounts of data—it just needs to change some pointers.

Databases use doubly linked lists to connect the so-called index leaf nodes. Each leaf node is stored in a database block or page; that is, the database's smallest storage unit. All index blocks are of the same size—typically a few kilobytes. The database uses the space in each block to the extent possible and stores as many index entries as possible in each block. That means that the index order is maintained on two different levels: the index entries within each leaf node, and the leaf nodes among each other using a doubly linked list.

Figure 1.1 Index Leaf Nodes and Corresponding Table Data

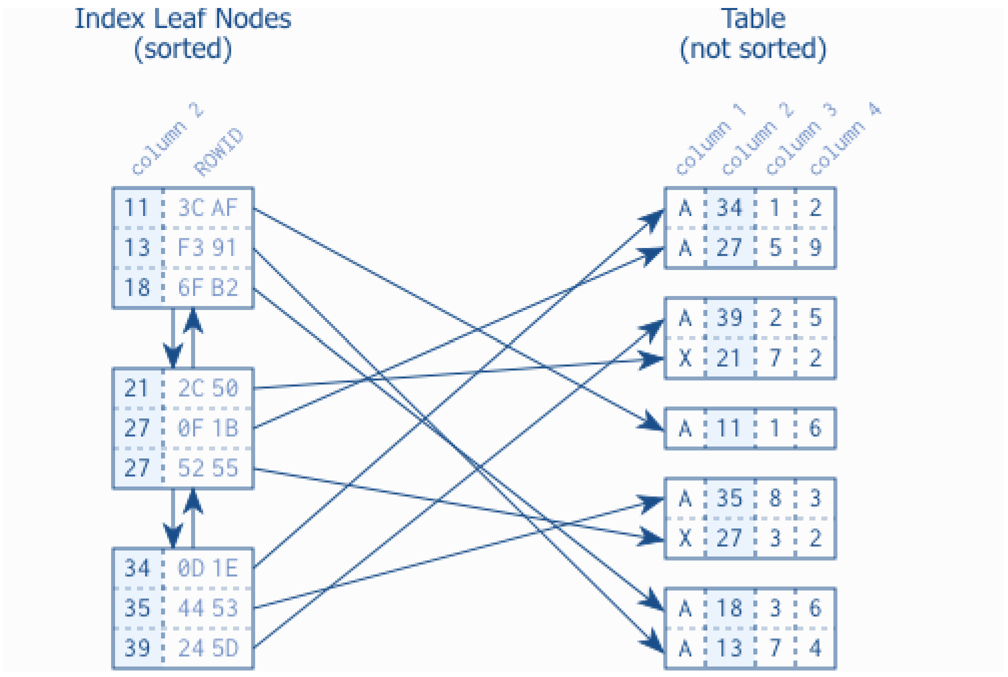


Figure 1.1 illustrates the index leaf nodes and their connection to the table data. Each index entry consists of the indexed columns (the key, column 2) and refers to the corresponding table row (via `ROWID` or `RID`). Unlike the index, the table data is stored in a heap structure and is not sorted at all. There is neither a relationship between the rows stored in the same table block nor is there any connection between the blocks.

1.3 BTREE

The index leaf nodes are stored in an arbitrary order—the position on the disk does not correspond to the logical position according to the index order. It is like a telephone directory with shuffled pages. If you search for “Smith” but first open the directory at “Robinson”, it is by no means granted that Smith follows Robinson. A database needs a second structure to find the entry among the shuffled pages quickly: a balanced search tree—in short: the B-tree.

Figure 1.2 B-tree Structure

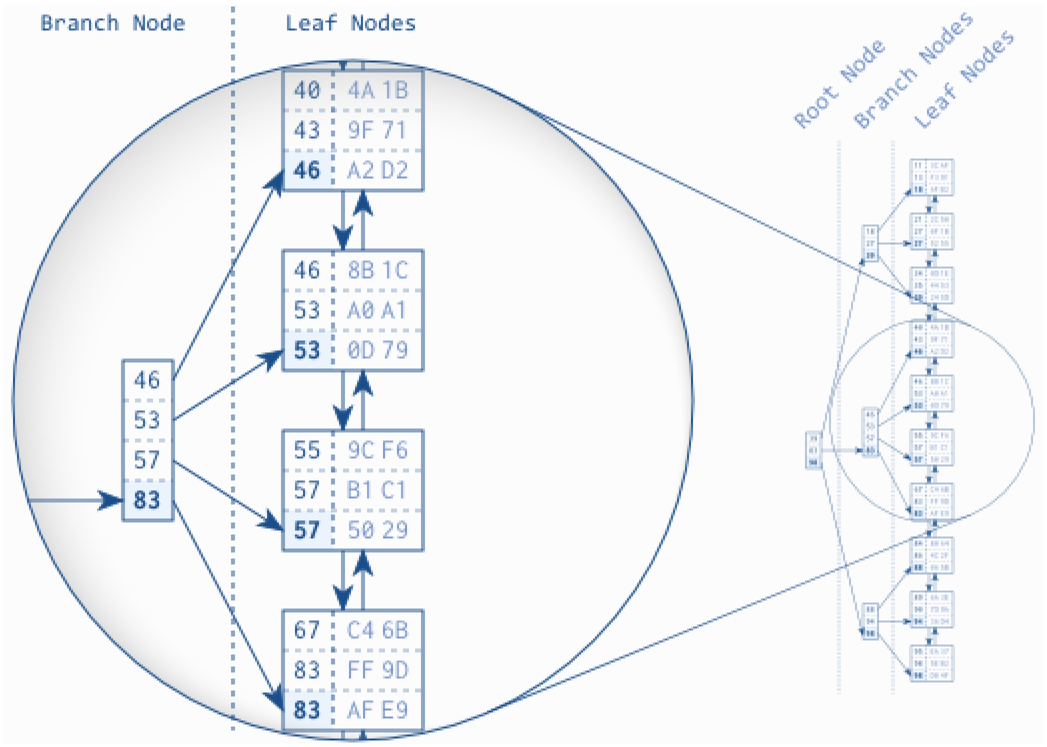


Figure 1.2 shows an example index with 30 entries. The doubly linked list establishes the logical order between the leaf nodes. The root and branch nodes support quick searching among the leaf nodes.

The figure highlights a branch node and the leaf nodes it refers to. Each branch node entry corresponds to the biggest value in the respective leaf node. Take the first leaf node as an example: the biggest value in this node is 46, which is thus stored in the corresponding branch node entry. The same is true for the other leaf nodes so that in the end the branch node has the values 46, 53, 57 and 83. According to this scheme, a branch layer is built up until all the leaf nodes are covered by a branch node.

The next layer is built similarly, but on top of the first branch node level. The procedure repeats until all keys fit into a single node, the *root node*. The structure is a *balanced search tree* because the tree depth is equal at every position; the distance between root node and leaf nodes is the same everywhere.

Once created, the database maintains the index automatically. It applies every `insert`, `delete` and `update` to the index and keeps the tree in balance, thus causing maintenance overhead for write operations.

Figure 1.3 B-Tree Traversal

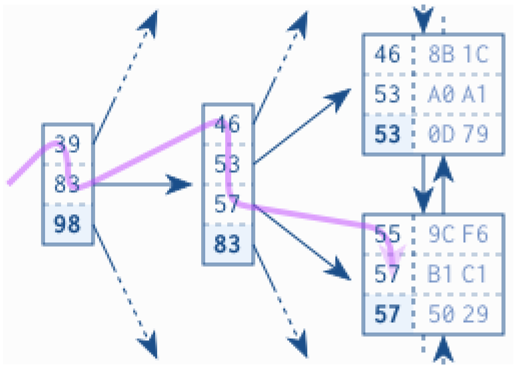


Figure 1.3 shows an index fragment to illustrate a search for the key “57”. The tree traversal starts at the root node on the left-hand side. Each entry is processed in ascending order until a value is greater than or equal to (>=) the search term (57). In the figure it is the entry 83. The database follows the reference to the corresponding branch node and repeats the procedure until the tree traversal reaches a leaf node.

The tree traversal is a very efficient operation—so efficient that I refer to it as the *first power of indexing*. It works almost instantly—even on a huge data set. That is primarily because of the tree balance, which allows accessing all elements with the same number of steps, and secondly because of the logarithmic growth of the tree depth. That means that the tree depth grows very slowly compared to the number of leaf nodes. Real world indexes with millions of records have a tree depth of four or five. A tree depth of six is hardly ever seen. The box *“Logarithmic Scalability”* describes this in more detail.

1.4 Slow Indexes

Despite the efficiency of the tree traversal, there are still cases where an index lookup doesn't work as fast as expected. This contradiction has fueled the myth of the *“degenerated index”* for a long time. The myth proclaims an index rebuild as the miracle solution. For now, you can take it for granted that rebuilding an index does not improve performance on the long run. The real reason trivial statements can be slow—even when using an index—can be explained on the basis of the previous sections.

The first ingredient for a slow index lookup is the leaf node chain. Consider the search for “57” in Figure 1.3 again. There are obviously two matching entries in the index. At least two entries are the same, to be more precise: the next leaf node could have further entries for “57”. The database *must* read the next leaf node to see if there are any more matching entries. That means that an index lookup not only needs to perform the tree traversal, it also needs to follow the leaf node chain.

The second ingredient for a slow index lookup is accessing the table. Even a single leaf node might contain many hits—often hundreds. The corresponding table data is usually scattered across many table blocks (see Figure 1.1“Index Leaf Nodes and Corresponding Table Data”). That means that there is an additional table access for each hit.

An index lookup requires three steps: (1) the tree traversal; (2) following the leaf node chain; (3) fetching the table data. The tree traversal is the only step that has an upper bound for the number of accessed blocks—the index depth. The other two steps might need to access many blocks—they cause a slow index lookup.

The origin of the “slow indexes” myth is the misbelief that an index lookup just traverses the tree, hence the idea that a slow index must be caused by a “broken” or “unbalanced” tree. The truth is that you can actually ask most databases how they use an index. The Oracle database is rather verbose in this respect and has three distinct operations that describe a basic index lookup:

INDEX UNIQUE SCAN

The `INDEX UNIQUE SCAN` performs the tree traversal only. The Oracle database uses this operation if a unique constraint ensures that the search criteria will match no more than one entry.

INDEX RANGE SCAN

The `INDEX RANGE SCAN` performs the tree traversal *and* follows the leaf node chain to find all matching entries. This is the fallback operation if multiple entries could possibly match the search criteria.

TABLE ACCESS BY INDEX ROWID

The `TABLE ACCESS BY INDEX ROWID` operation retrieves the row from the table. This operation is (often) performed for every matched record from a preceding index scan operation.

The important point is that an `INDEX RANGE SCAN` can potentially read a large part of an index. If there is one more table access for each row, the query can become slow even when using an index.

1.5 Clustered Index / Non-Clustered Index

A clustered index (SQL Server, MySQL/InnoDB) is a table stored in an index B-Tree structure. There is no second data structure (heap-table) for the table.

A non-clustered index is an index that refers to another data structure containing further table columns.

Accessing table data via a secondary index (index on a clustered index) is slower than a similar query on a heap-table.

SQL Server supports clustered index optionally. You have a free choice between clustered indexes and heap-tables. There can be at most one clustered index per table. Dropping a clustered index transforms the table into a heap-table. Adding a clustered index to a heap table actually drops the heap structure. SQL Server supports non-unique clustered indexes on arbitrary columns. Creating an SQL Server table without clustering index requires the use of the `NONCLUSTERED` clause:

```
CREATE TABLE (
  id    NUMBER NOT NULL,
  [...]
  CONSTRAINT pk PRIMARY KEY NONCLUSTERED (id)
)
```

The MySQL InnoDB engine has mandatory clustered indexes. That means there is always a clustered index, often using the primary key. If there is no suitable unique key available, MySQL will use a generated row ID for that purpose. The MyISAM storage engine doesn't support clustered indexes and uses heap-tables all the time.

The Oracle database has optional clustered indexes called Index-Organized Tables. They work on the primary key only.

1.6 MYTH: INDEXES CAN DEGENERATE

The most prominent myth is that an index can become degenerated after a while and must be re-built regularly. First of all, the database keeps the tree balance—*always*. It is not possible that a single fragment of the index grows deeper and deeper until the tree traversal becomes slow. What can happen is that the index become bigger than needed. If there are many `UPDATE` or `DELETE` statements involved space utilization can become suboptimal. However, even if the index is bigger than required, it is very unlikely that the depth of the index grows because of that. The number of entries in the index must typically grow by a factor of hundred to increase the index depth by one level.

Rebuilding an index might reduce the number of leaf nodes by about 20% - 30%. The most you can possibly expect from this reduction is 20%-30% for very expensive operations like a `FULL INDEX SCAN`. The typical `INDEX UNIQUE SCAN` gain of an index rebuild is 0%-2% because the depth of the index is not reduced by the rebuild.

2 VIEWS

A database view is a searchable object in a database that is defined by a query. Though a view doesn't store data, some refer to a views as "virtual tables," you can query a view like you can a table. A view can combine data from two or more table, using joins, and also just contain a subset of information. This makes them convenient to abstract, or hide, complicated queries.

Below is a visual depiction of a view:

2.1 HOW TO CREATE AND USE A DATABASE VIEW

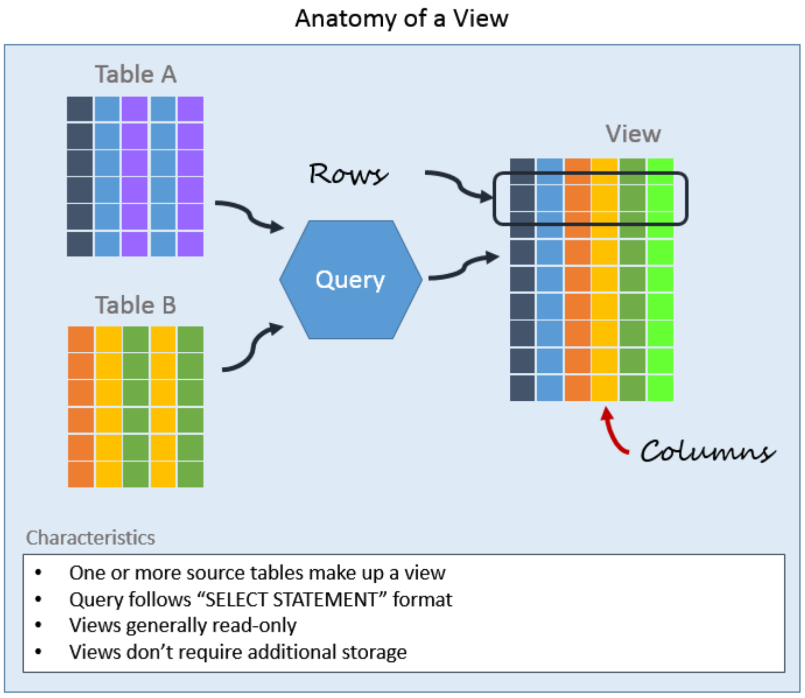
A view is created from a query using the CREATE VIEW command. In the example below we are creating a PopularBooks view based of a query which selects all Books that have the IsPopular field checked. The Query is colored in Blue.

```
CREATE VIEW PopularBooks AS
SELECT ISBN, Title, Author,
PublishDate
FROM Books
WHERE IsPopular = 1
```

Once a view is created you can used then as you would any table in a SELECT statement. For example, to list all the popular book titles ordered by author you could write:

```
SELECT Author, Title
FROM PopularBooks
ORDER BY Author
```

In general you can use any of the SELECT clauses, such as GROUP BY, in a select statement containing a view.



- One or more source tables make up a view
- Query follows "SELECT STATEMENT" format
- Views generally read-only
- Views don't require additional storage

2.2 BENEFITS OF A DATABASE VIEW

There are many benefits to using views. Listed below are some of the one that come to mind:

- **Enforce Business Rules** – Use views to define business rules, such as when an items is active, or what is meant by "popular." By placing complicated or misunderstood business logic into the view, you can be sure to present a unified portrayal of the data. This increases use and quality.
- **Consistency** – Simplify complicated query logic and calculations by hiding it behind the view's definition. Once defined they calculations are reference from the view rather than being restated in separate queries. This makes for less mistakes and easier maintenance of code.

- **Security** – Restrict access to a table, yet allow users to access non-confidential data via views. For example, you can restrict access to the employee table, that contains social security numbers, but allow access to a view containing name and phone number.
- **Simplicity** – Databases with many tables possess complex relationships, which can be difficult to navigate if you aren't comfortable using Joins. Use views to provide a "flattened" view of the database for reporting or ad-hoc queries.
- **Space** – Views take up very little space, as the data is stored once in the source table. Some DBMS all you to create an index on a view, so in some cases views do take up more space than the definition.

2.3 Disadvantages of Views

- **Performance** – What may seem like a simple query against a view could turn out to be a hugely complex job for the database engine. That is because each time a view is referenced, the query used to define it, is rerun.
- **Modifications** – Not all views support INSERT, UPDATE, or DELETE operations. In general, in order to support these operations, the primary key and required fields must be present in the view. Complex multi-table views are generally read only.