COVERS ALL MAJOR SQL DATABASES

SQL PERFORMANCE EXPLAINED

ENGLISH EDITION

EVERYTHING DEVELOPERS NEED TO KNOW ABOUT SQL PERFORMANCE

MARKUS WINAND

Publisher:
Markus Winand

Maderspergerstasse 1-3/9/11
1160 Wien
AUSTRIA
<office@winand.at>

DGS - Druck- u. Graphikservice GmbH — Wien — Austria

Cover design:
tomasio.design — Mag. Thomas Weninger — Wien — Austria

Cover photo:
Brian Arnold — Turriff — UK

Copy editor:
Nathan Ingvalson — Graz — Austria

2013-10-03

SQL Performance Explained

Everything developers need to know about SQL performance

Markus Winand
Vienna, Austria

Contents

DEVELOPERS NEED TO INDEX

SQL performance problems are as old as SQL itself—some might even say that SQL is inherently slow. Although this might have been true in the early days of SQL, it is definitely not true anymore. Nevertheless SQL performance problems are still commonplace. How does this happen?

The SQL language is perhaps the most successful fourth-generation programming language (4GL). Its main benefit is the capability to separate *"what"* and *"how"*. An SQL statement is a straight description *what* is needed without instructions as to *how* to get it done. Consider the following example:

```
SELECT date_of_birth
  FROM employees
 WHERE last_name = 'WINAND'
```

The SQL query reads like an English sentence that explains the requested data. Writing SQL statements generally does not require any knowledge about inner workings of the database or the storage system (such as disks, files, etc.). There is no need to tell the database which files to open or how to find the requested rows. Many developers have years of SQL experience yet they know very little about the processing that happens in the database.

The separation of concerns—what is needed versus how to get it—works remarkably well in SQL, but it is still not perfect. The abstraction reaches its limits when it comes to performance: the author of an SQL statement by definition does not care *how* the database executes the statement. Consequently, the author is not responsible for slow execution. However, experience proves the opposite; i.e., the author must know a little bit about the database to prevent performance problems.

It turns out that the only thing *developers* need to learn is how to index. Database indexing is, in fact, a development task. That is because the most important information for proper indexing is not the storage system configuration or the hardware setup. The most important information for indexing is how the application queries the data. This knowledge—about

the access path—is not very accessible to database administrators (DBAs) or external consultants. Quite some time is needed to gather this information through reverse engineering of the application: development, on the other hand, has that information anyway.

This book covers everything developers need to know about indexes—and nothing more. To be more precise, the book covers the most important index type only: the *B-tree index*.

The B-tree index works almost identically in many databases. The book only uses the terminology of the Oracle® database, but the principles apply to other databases as well. Side notes provide relevant information for MySQL, PostgreSQL and SQL Server®.

The structure of the book is tailor-made for developers; most chapters correspond to a particular part of an SQL statement.

CHAPTER 1 - Anatomy of an Index
: The first chapter is the only one that doesn't cover SQL specifically; it is about the fundamental structure of an index. An understanding of the index structure is essential to following the later chapters—don't skip this!

  Although the chapter is rather short—only about eight pages—after working through the chapter you will already understand the phenomenon of slow indexes.

CHAPTER 2 - The Where Clause
: This is where we pull out all the stops. This chapter explains all aspects of the **`where`** clause, from very simple single column lookups to complex clauses for ranges and special cases such as `LIKE`.

  This chapter makes up the main body of the book. Once you learn to use these techniques, you will write much faster SQL.

CHAPTER 3 - Performance and Scalability
: This chapter is a little digression about performance measurements and database scalability. See why adding hardware is not the best solution to slow queries.

CHAPTER 4 - The Join Operation
: Back to SQL: here you will find an explanation of how to use indexes to perform a fast table join.

CHAPTER 5 - Clustering Data

Have you ever wondered if there is any difference between selecting a single column or all columns? Here is the answer—along with a trick to get even better performance.

CHAPTER 6 - Sorting and Grouping

Even **`order by`** and **`group by`** can use indexes.

CHAPTER 7 - Partial Results

This chapter explains how to benefit from a "pipelined" execution if you don't need the full result set.

CHAPTER 8 - Insert, Delete and Update

How do indexes affect write performance? Indexes don't come for free—use them wisely!

APPENDIX A - Execution Plans

Asking the database how it executes a statement.

Chapter 1

Anatomy of an Index

"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.

Clustered Indexes

SQL Server and MySQL (using InnoDB) take a broader view of what "*index*" means. They refer to tables that consist of the index structure only as *clustered indexes*. These tables are called Index-Organized Tables (IOT) in the Oracle database.

Chapter 5, "*Clustering Data*", describes them in more detail and explains their advantages and disadvantages.

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 entries 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.

The Index 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.

Doubly linked lists are also used for collections (containers) in many programming languages.

Programming Language	Name
Java	java.util.LinkedList
.NET Framework	System.Collections.Generic.LinkedList
C++	std::list

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

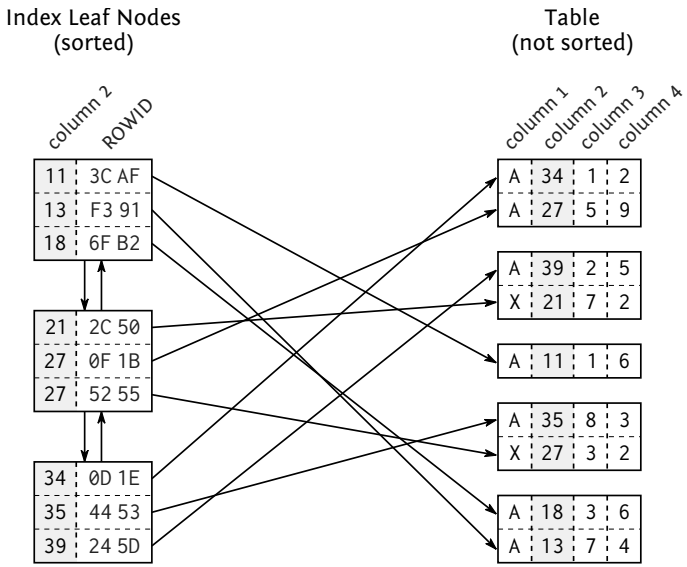


Figure 1.1 illustrates the index leaf nodes and their connection to the table data. Each index entry consists of the indexed columns (the key, column 2) and refers to the corresponding table row (via `ROWID` or `RID`). Unlike the index, the table data is stored in a heap structure and is not sorted at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

The Search Tree (B-Tree)

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

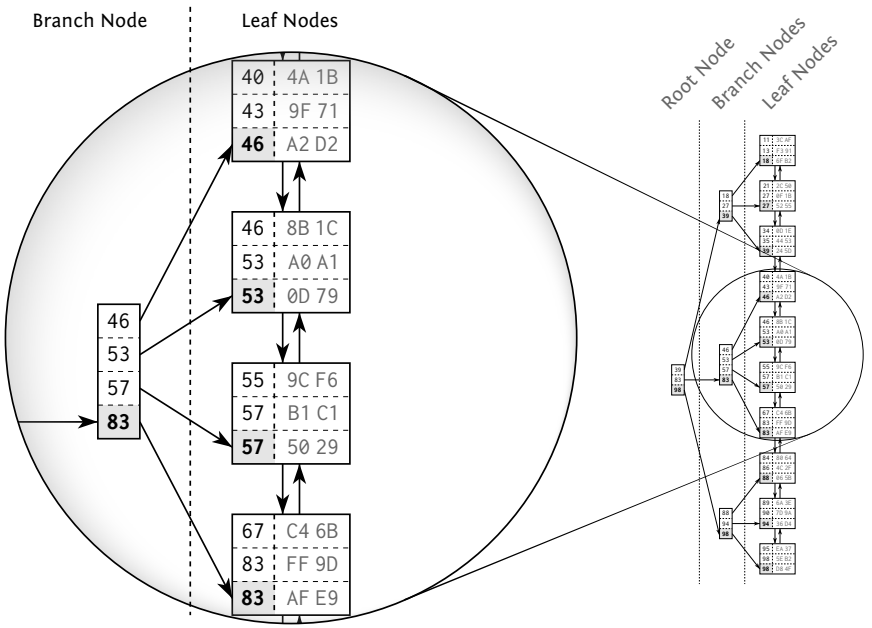


Figure 1.2 shows an example index with 30 entries. The doubly linked list establishes the logical order between the leaf nodes. The root and branch nodes support quick searching among the leaf nodes.

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. That is, 46 in the first leaf node so that the first branch node entry is also 46. 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.

Note

A B-tree is a balanced tree—not a binary tree.

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. Chapter 8, "*Modifying Data*", explains this in more detail.

Figure 1.3. B-Tree Traversal

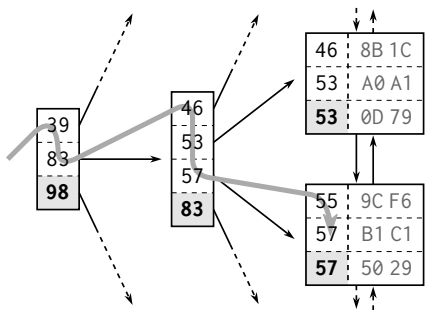


Figure 1.3 shows an index fragment to illustrate a search for the key "57". The tree traversal starts at the root node on the left-hand side. Each entry is processed in ascending order until a value is greater than or equal to (>=) the search term (57). In the figure it is the entry 83. The database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

Important

The B-tree enables the database to find a leaf node quickly.

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.

Slow Indexes, Part I

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. 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.

Logarithmic Scalability

In mathematics, the logarithm of a number to a given base is the power or exponent to which the base must be raised in order to produce the number [Wikipedia[1]].

In a search tree the base corresponds to the number of entries per branch node and the exponent to the tree depth. The example index in Figure 1.2 holds up to four entries per node and has a tree depth of three. That means that the index can hold up to 64 (4^3) entries. If it grows by one level, it can already hold 256 entries (4^4). Each time a level is *added*, the maximum number of index entries *quadruples*. The logarithm reverses this function. The tree depth is therefore $\log_4$(number-of-index-entries).

The logarithmic growth enables the example index to search a million records with ten tree levels, but a real world index is even more efficient. The main factor that affects the tree depth, and therefore the lookup performance, is the number of entries in each tree node. This number corresponds to—mathematically speaking—the basis of the logarithm. The higher the basis, the shallower the tree, the faster the traversal.

Tree Depth	Index Entries
3	64
4	256
5	1,024
6	4,096
7	16,384
8	65,536
9	262,144
10	1,048,576

Databases exploit this concept to a maximum extent and put as many entries as possible into each node—often hundreds. That means that every new index level supports a hundred times more entries.

[1] http://en.wikipedia.org/wiki/Logarithm

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.

Chapter 2

The Where Clause

The previous chapter described the structure of indexes and explained the cause of poor index performance. In the next step we learn how to spot and avoid these problems in SQL statements. We start by looking at the **`where`** clause.

The **`where`** clause defines the search condition of an SQL statement, and it thus falls into the core functional domain of an index: finding data quickly. Although the **`where`** clause has a huge impact on performance, it is often phrased carelessly so that the database has to scan a large part of the index. The result: a poorly written **`where`** clause is the first ingredient of a slow query.

This chapter explains how different operators affect index usage and how to make sure that an index is usable for as many queries as possible. The last section shows common anti-patterns and presents alternatives that deliver better performance.

The Equality Operator

The equality operator is both the most trivial and the most frequently used SQL operator. Indexing mistakes that affect performance are still very common and **`where`** clauses that combine multiple conditions are particularly vulnerable.

This section shows how to verify index usage and explains how concatenated indexes can optimize combined conditions. To aid understanding, we will analyze a slow query to see the real world impact of the causes explained in Chapter 1.

PRIMARY KEYS

We start with the simplest yet most common **where** clause: the primary key lookup. For the examples throughout this chapter we use the EMPLOYEES table defined as follows:

```
CREATE TABLE employees (
   employee_id   NUMBER         NOT NULL,
   first_name    VARCHAR2(1000) NOT NULL,
   last_name     VARCHAR2(1000) NOT NULL,
   date_of_birth DATE           NOT NULL,
   phone_number  VARCHAR2(1000) NOT NULL,
   CONSTRAINT employees_pk PRIMARY KEY (employee_id)
);
```

The database automatically creates an index for the primary key. That means there is an index on the EMPLOYEE_ID column, even though there is no **create index** statement.

The following query uses the primary key to retrieve an employee's name:

```
SELECT first_name, last_name
  FROM employees
 WHERE employee_id = 123
```

The **where** clause cannot match multiple rows because the primary key constraint ensures uniqueness of the EMPLOYEE_ID values. The database does not need to follow the index leaf nodes—it is enough to traverse the index tree. We can use the so-called *execution plan* for verification:

```
------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |    2 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |    2 |
|*2 |  INDEX UNIQUE SCAN          | EMPLOYEES_PK |    1 |    1 |
------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"=123)
```

The Oracle execution plan shows an INDEX UNIQUE SCAN—the operation that only traverses the index tree. It fully utilizes the logarithmic scalability of the index to find the entry very quickly—almost independent of the table size.

Tip

The *execution plan* (sometimes *explain plan* or *query plan*) shows the steps the database takes to execute an SQL statement. Appendix A on page 165 explains how to retrieve and read execution plans with other databases.

After accessing the index, the database must do one more step to fetch the queried data (FIRST_NAME, LAST_NAME) from the table storage: the TABLE ACCESS BY INDEX ROWID operation. This operation can become a performance bottleneck—as explained in "Slow Indexes, Part I"—but there is no such risk in connection with an INDEX UNIQUE SCAN. This operation cannot deliver more than one entry so it cannot trigger more than one table access. That means that the ingredients of a slow query are not present with an INDEX UNIQUE SCAN.

Primary Keys without Unique Index

A primary key does not necessarily need a unique index—you can use a non-unique index as well. In that case the Oracle database does not use an INDEX UNIQUE SCAN but instead the INDEX RANGE SCAN operation. Nonetheless, the constraint still maintains the uniqueness of keys so that the index lookup delivers at most one entry.

One of the reasons for using non-unique indexes for a primary keys are *deferrable constraints*. As opposed to regular constraints, which are validated during statement execution, the database postpones the validation of deferrable constraints until the transaction is committed. Deferred constraints are required for inserting data into tables with circular dependencies.

CONCATENATED INDEXES

Even though the database creates the index for the primary key automatically, there is still room for manual refinements if the key consists of multiple columns. In that case the database creates an index on all primary key columns—a so-called *concatenated* index (also known as *multi-column*, *composite* or *combined* index). Note that the column order of a concatenated index has great impact on its usability so it must be chosen carefully.

For the sake of demonstration, let's assume there is a company merger. The employees of the other company are added to our `EMPLOYEES` table so it becomes ten times as large. There is only one problem: the `EMPLOYEE_ID` is not unique across both companies. We need to extend the primary key by an extra identifier—e.g., a subsidiary ID. Thus the new primary key has two columns: the `EMPLOYEE_ID` as before and the `SUBSIDIARY_ID` to reestablish uniqueness.

The index for the new primary key is therefore defined in the following way:

```
CREATE UNIQUE INDEX employee_pk
    ON employees (employee_id, subsidiary_id);
```

A query for a particular employee has to take the full primary key into account—that is, the `SUBSIDIARY_ID` column also has to be used:

```
SELECT first_name, last_name
  FROM employees
 WHERE employee_id   = 123
   AND subsidiary_id = 30
```

```
-------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |    2 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |    2 |
|*2 |  INDEX UNIQUE SCAN          | EMPLOYEES_PK |    1 |    1 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPLOYEE_ID"=123 AND "SUBSIDIARY_ID"=30)
```

Whenever a query uses the complete primary key, the database can use an INDEX UNIQUE SCAN—no matter how many columns the index has. But what happens when using only one of the key columns, for example, when searching all employees of a subsidiary?

```
SELECT first_name, last_name
  FROM employees
 WHERE subsidiary_id = 20
```

```
-----------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
-----------------------------------------------------
|  0 | SELECT STATEMENT  |           |  106 |  478 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |  106 |  478 |
-----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("SUBSIDIARY_ID"=20)
```

The execution plan reveals that the database does not use the index. Instead it performs a TABLE ACCESS FULL. As a result the database reads the entire table and evaluates every row against the **where** clause. The execution time grows with the table size: if the table grows tenfold, the TABLE ACCESS FULL takes ten times as long. The danger of this operation is that it is often fast enough in a small development environment, but it causes serious performance problems in production.

Full Table Scan

The operation TABLE ACCESS FULL, also known as *full table scan*, can be the most efficient operation in some cases anyway, in particular when retrieving a large part of the table.

This is partly due to the overhead for the index lookup itself, which does not happen for a TABLE ACCESS FULL operation. This is mostly because an index lookup reads one block after the other as the database does not know which block to read next until the current block has been processed. A FULL TABLE SCAN must get the entire table anyway so that the database can read larger chunks at a time (*multi block read*). Although the database reads more data, it might need to execute fewer read operations.

The database does not use the index because it cannot use single columns from a concatenated index arbitrarily. A closer look at the index structure makes this clear.

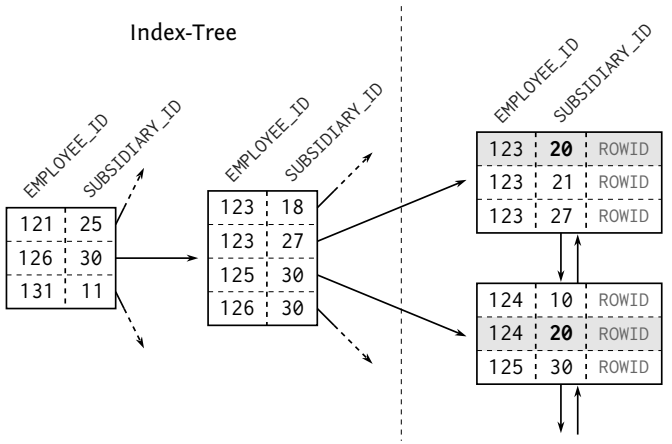
A concatenated index is just a B-tree index like any other that keeps the indexed data in a sorted list. The database considers each column according to its position in the index definition to sort the index entries. The first column is the primary sort criterion and the second column determines the order only if two entries have the same value in the first column and so on.

Important

A concatenated index is *one index across multiple columns*.

The ordering of a two-column index is therefore like the ordering of a telephone directory: it is first sorted by surname, then by first name. That means that a two-column index does not support searching on the second column alone; that would be like searching a telephone directory by first name.

Figure 2.1. Concatenated Index



The index excerpt in Figure 2.1 shows that the entries for subsidiary 20 are not stored next to each other. It is also apparent that there are no entries with `SUBSIDIARY_ID = 20` in the tree, although they exist in the leaf nodes. The tree is therefore useless for this query.

Tip

Visualizing an index helps in understanding what queries the index supports. You can query the database to retrieve the entries in index order (SQL:2008 syntax, see page 144 for proprietary solutions using LIMIT, TOP or ROWNUM):

```
SELECT <INDEX COLUMN LIST>
  FROM <TABLE>
 ORDER BY <INDEX COLUMN LIST>
 FETCH FIRST 100 ROWS ONLY;
```

If you put the index definition and table name into the query, you will get a sample from the index. Ask yourself if the requested rows are clustered in a central place. If not, the index tree cannot help find that place.

We could, of course, add another index on SUBSIDIARY_ID to improve query speed. There is however a better solution—at least if we assume that searching on EMPLOYEE_ID alone does not make sense.

We can take advantage of the fact that the first index column is always usable for searching. Again, it is like a telephone directory: you don't need to know the first name to search by last name. The trick is to reverse the index column order so that the SUBSIDIARY_ID is in the first position:

```
CREATE UNIQUE INDEX EMPLOYEES_PK
    ON EMPLOYEES (SUBSIDIARY_ID, EMPLOYEE_ID);
```

Both columns together are still unique so queries with the full primary key can still use an INDEX UNIQUE SCAN but the sequence of index entries is entirely different. The SUBSIDIARY_ID has become the primary sort criterion. That means that all entries for a subsidiary are in the index consecutively so the database can use the B-tree to find their location.

IMPORTANT

The most important consideration when defining a concatenated index is how to choose the column order so it can support as many SQL queries as possible.

The execution plan confirms that the database uses the "reversed" index. The `SUBSIDIARY_ID` alone is not unique anymore so the database must follow the leaf nodes in order to find all matching entries: it is therefore using the `INDEX RANGE SCAN` operation.

```
-------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |  106 |   75 |
| 1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |  106 |   75 |
|*2 |  INDEX RANGE SCAN           | EMPLOYEE_PK |  106 |    2 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("SUBSIDIARY_ID"=20)
```

In general, a database can use a concatenated index when searching with the leading (leftmost) columns. An index with three columns can be used when searching for the first column, when searching with the first two columns together, and when searching using all columns.

Even though the two-index solution delivers very good **`select`** performance as well, the single-index solution is preferable. It not only saves storage space, but also the maintenance overhead for the second index. The fewer indexes a table has, the better the **`insert`**, **`delete`** and **`update`** performance.

To define an optimal index you must understand more than just how indexes work—you must also know how the application queries the data. This means you have to know the column combinations that appear in the **`where`** clause.

Defining an optimal index is therefore very difficult for external consultants because they don't have an overview of the application's access paths. Consultants can usually consider one query only. They do not exploit the extra benefit the index could bring for other queries. Database administrators are in a similar position as they might know the database schema but do not have deep insight into the access paths.

The only place where the technical database knowledge meets the functional knowledge of the business domain is the development department. Developers have a feeling for the data and know the access path. They can properly index to get the best benefit for the overall application without much effort.

Slow Indexes, Part II

The previous section explained how to gain additional benefits from an existing index by changing its column order, but the example considered only two SQL statements. Changing an index, however, may affect all queries on the indexed table. This section explains the way databases pick an index and demonstrates the possible side effects when changing existing indexes.

The adopted `EMPLOYEE_PK` index improves the performance of all queries that search by subsidiary only. It is however usable for all queries that search by `SUBSIDIARY_ID`—regardless of whether there are any additional search criteria. That means the index becomes usable for queries that used to use another index with another part of the **`where`** clause. In that case, if there are multiple access paths available it is the optimizer's job to choose the best one.

The Query Optimizer

The query optimizer, or query planner, is the database component that transforms an SQL statement into an execution plan. This process is also called *compiling* or *parsing*. There are two distinct optimizer types.

Cost-based optimizers (CBO) generate many execution plan variations and calculate a *cost* value for each plan. The cost calculation is based on the operations in use and the estimated row numbers. In the end the cost value serves as the benchmark for picking the "best" execution plan.

Rule-based optimizers (RBO) generate the execution plan using a hard-coded rule set. Rule based optimizers are less flexible and are seldom used today.

Changing an index might have unpleasant side effects as well. In our example, it is the internal telephone directory application that has become very slow since the merger. The first analysis identified the following query as the cause for the slowdown:

```
SELECT first_name, last_name, subsidiary_id, phone_number
  FROM employees
 WHERE last_name  = 'WINAND'
   AND subsidiary_id = 30
```

The execution plan is:

Example 2.1. Execution Plan with Revised Primary Key Index

```
-------------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |   30 |
|*1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |   30 |
|*2 |  INDEX RANGE SCAN           | EMPLOYEES_PK |   40 |    2 |
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - filter("LAST_NAME"='WINAND')
  2 - access("SUBSIDIARY_ID"=30)
```

The execution plan uses an index and has an overall cost value of 30. So far, so good. It is however suspicious that it uses the index we just changed—that is enough reason to suspect that our index change caused the performance problem, especially when bearing the old index definition in mind—it started with the `EMPLOYEE_ID` column which is not part of the **`where`** clause at all. The query could not use that index before.

For further analysis, it would be nice to compare the execution plan before and after the change. To get the original execution plan, we could just deploy the old index definition again, however most databases offer a simpler method to prevent using an index for a specific query. The following example uses an Oracle *optimizer hint* for that purpose.

```
SELECT /*+ NO_INDEX(EMPLOYEES EMPLOYEE_PK) */
       first_name, last_name, subsidiary_id, phone_number
  FROM employees
 WHERE last_name  = 'WINAND'
   AND subsidiary_id = 30
```

The execution plan that was presumably used before the index change did not use an index at all:

```
-------------------------------------------------
| Id | Operation         | Name      | Rows | Cost |
-------------------------------------------------
|  0 | SELECT STATEMENT  |           |    1 |  477 |
|* 1 |  TABLE ACCESS FULL| EMPLOYEES |    1 |  477 |
-------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("LAST_NAME"='WINAND' AND "SUBSIDIARY_ID"=30)
```

Even though the TABLE ACCESS FULL must read and process the entire table, it seems to be faster than using the index in this case. That is particularly unusual because the query matches one row only. Using an index to find a single row should be much faster than a full table scan, but in this case it is not. The index seems to be slow.

In such cases it is best to go through each step of the troublesome execution plan. The first step is the INDEX RANGE SCAN on the EMPLOYEES_PK index. That index does not cover the LAST_NAME column—the INDEX RANGE SCAN can consider the SUBSIDIARY_ID filter only; the Oracle database shows this in the "Predicate Information" area—entry "2" of the execution plan. There you can see the conditions that are applied for each operation.

TIP

Appendix A, "*Execution Plans*", explains how to find the "Predicate Information" for other databases.

The INDEX RANGE SCAN with operation ID 2 (Example 2.1 on page 19) applies only the SUBSIDIARY_ID=30 filter. That means that it traverses the index tree to find the first entry for SUBSIDIARY_ID 30. Next it follows the leaf node chain to find all other entries for that subsidiary. The result of the INDEX ONLY SCAN is a list of ROWIDs that fulfill the SUBSIDIARY_ID condition: depending on the subsidiary size, there might be just a few ones or there could be many hundreds.

The next step is the TABLE ACCESS BY INDEX ROWID operation. It uses the ROWIDs from the previous step to fetch the rows—all columns—from the table. Once the LAST_NAME column is available, the database can evaluate the remaining part of the **where** clause. That means the database has to fetch all rows for SUBSIDIARY_ID=30 before it can apply the LAST_NAME filter.

The statement's response time does not depend on the result set size but on the number of employees in the particular subsidiary. If the subsidiary has just a few members, the `INDEX RANGE SCAN` provides better performance. Nonetheless a `TABLE ACCESS FULL` can be faster for a huge subsidiary because it can read large parts from the table in one shot (see "Full Table Scan" on page 13).

The query is slow because the index lookup returns many `ROWIDs`—one for each employee of the original company—and the database must fetch them individually. It is the perfect combination of the two ingredients that make an index slow: the database reads a wide index range and has to fetch many rows individually.

Choosing the best execution plan depends on the table's data distribution as well so the optimizer uses statistics about the contents of the database. In our example, a histogram containing the distribution of employees over subsidiaries is used. This allows the optimizer to estimate the number of rows returned from the index lookup—the result is used for the cost calculation.

Statistics

A cost-based optimizer uses statistics about tables, columns, and indexes. Most statistics are collected on the column level: the number of distinct values, the smallest and largest values (data range), the number of `NULL` occurrences and the column histogram (data distribution). The most important statistical value for a table is its size (in rows and blocks).

The most important index statistics are the tree depth, the number of leaf nodes, the number of distinct keys and the clustering factor (see Chapter 5, "*Clustering Data*").

The optimizer uses these values to estimate the selectivity of the **`where`** clause predicates.

If there are no statistics available—for example because they were deleted—the optimizer uses default values. The default statistics of the Oracle database suggest a small index with medium selectivity. They lead to the estimate that the INDEX RANGE SCAN will return 40 rows. The execution plan shows this estimation in the Rows column (again, see Example 2.1 on page 19). Obviously this is a gross underestimate, as there are 1000 employees working for this subsidiary.

If we provide correct statistics, the optimizer does a better job. The following execution plan shows the new estimation: 1000 rows for the INDEX RANGE SCAN. Consequently it calculated a higher cost value for the subsequent table access.

```
-------------------------------------------------------------
|Id |Operation                    | Name         | Rows | Cost |
-------------------------------------------------------------
| 0 |SELECT STATEMENT             |              |    1 |  680 |
|*1 | TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |    1 |  680 |
|*2 |  INDEX RANGE SCAN           | EMPLOYEES_PK | 1000 |    4 |
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  1 - filter("LAST_NAME"='WINAND')
  2 - access("SUBSIDIARY_ID"=30)
```

The cost value of 680 is even higher than the cost value for the execution plan using the FULL TABLE SCAN (477, see page 20). The optimizer will therefore automatically prefer the FULL TABLE SCAN.

This example of a slow index should not hide the fact that proper indexing is the best solution. Of course searching on last name is best supported by an index on LAST_NAME:

```
CREATE INDEX emp_name ON employees (last_name);;
```

Using the new index, the optimizer calculates a cost value of 3:

Example 2.2. Execution Plan with Dedicated Index

```
-----------------------------------------------------------------
| Id | Operation                    | Name      | Rows | Cost |
-----------------------------------------------------------------
|  0 | SELECT STATEMENT             |           |    1 |    3 |
|* 1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES |    1 |    3 |
|* 2 |   INDEX RANGE SCAN           | EMP_NAME  |    1 |    1 |
-----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("SUBSIDIARY_ID"=30)
   2 - access("LAST_NAME"='WINAND')
```

The index access delivers—according to the optimizer's estimation—one row only. The database thus has to fetch only that row from the table: this is definitely faster than a `FULL TABLE SCAN`. A properly defined index is still better than the original full table scan.

The two execution plans from Example 2.1 (page 19) and Example 2.2 are almost identical. The database performs the same operations and the optimizer calculated similar cost values, nevertheless the second plan performs much better. The efficiency of an `INDEX RANGE SCAN` may vary over a wide range—especially when followed by a table access. Using an index does not automatically mean a statement is executed in the best way possible.

SQL Performance Explained

SQL Performance Explained helps developers to improve database performance. The focus is on SQL—it covers all major SQL databases without getting lost in the details of any one specific product.

Starting with the basics of indexing and the `where` clause, SQL Performance Explained guides developers through all parts of an SQL statement and explains the pitfalls of object-relational mapping (ORM) tools like Hibernate.

Topics covered include:

- Using multi-column indexes
- Correctly applying SQL functions
- Efficient use of `LIKE` queries
- Optimizing join operations
- Clustering data to improve performance
- Pipelined execution of `order by` and `group by`
- Getting the best performance for pagination queries
- Understanding the scalability of databases

Its systematic structure makes SQL Performance Explained both a textbook and a reference manual that should be on every developer's bookshelf.

Covers

✓ Oracle® Database ✓ SQL Server® ✓ MySQL ✓ PostgreSQL

About Markus Winand

Markus Winand has been developing SQL applications since 1998. His main interests include performance, scalability, reliability, and generally all other technical aspects of software quality. Markus currently works as an independent trainer and coach in Vienna, Austria. http://winand.at/

ISBN 978-3-9503078-2-5

EUR 29.95
GBP 26.99

9 783950 307825 >