1/30   8:20:17   ✳✳✳

← → →

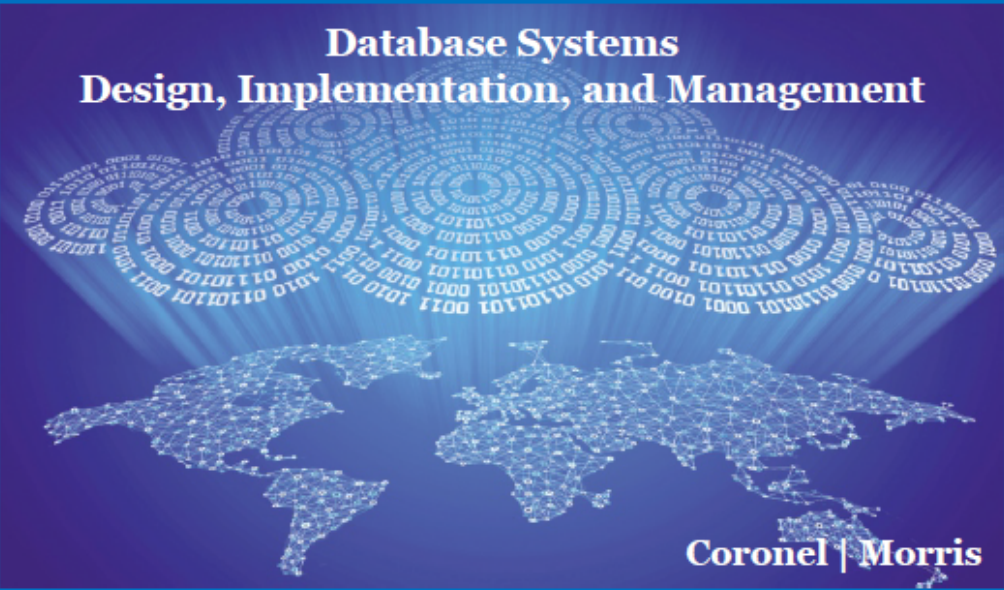# Perf. tuning

We tune sports cars – so why not tune DBs?!

Ch.11

11e

**Database Systems
Design, Implementation, and Management**

Coronel | Morris

Chapter 11

Database Performance Tuning and Query
Optimization

'Tuning'

Database Performance-Tuning Concepts

- Goal of database performance is to execute queries as fast as possible

- **Database performance tuning**: Set of activities and procedures that reduce response time of database system

- Fine-tuning the performance of a system requires that all factors must operate at optimum level with minimal bottlenecks

Performance Tuning: Client and Server

- Client side
  - **SQL performance tuning**: Generates SQL query that returns correct answer in least amount of time
    - Using minimum amount of resources at server
- Server side
  - **DBMS performance tuning:** DBMS environment configured to respond to clients' requests as fast as possible
    - Optimum use of existing resources

Figure 11.1 - Basic DBMS Architecture
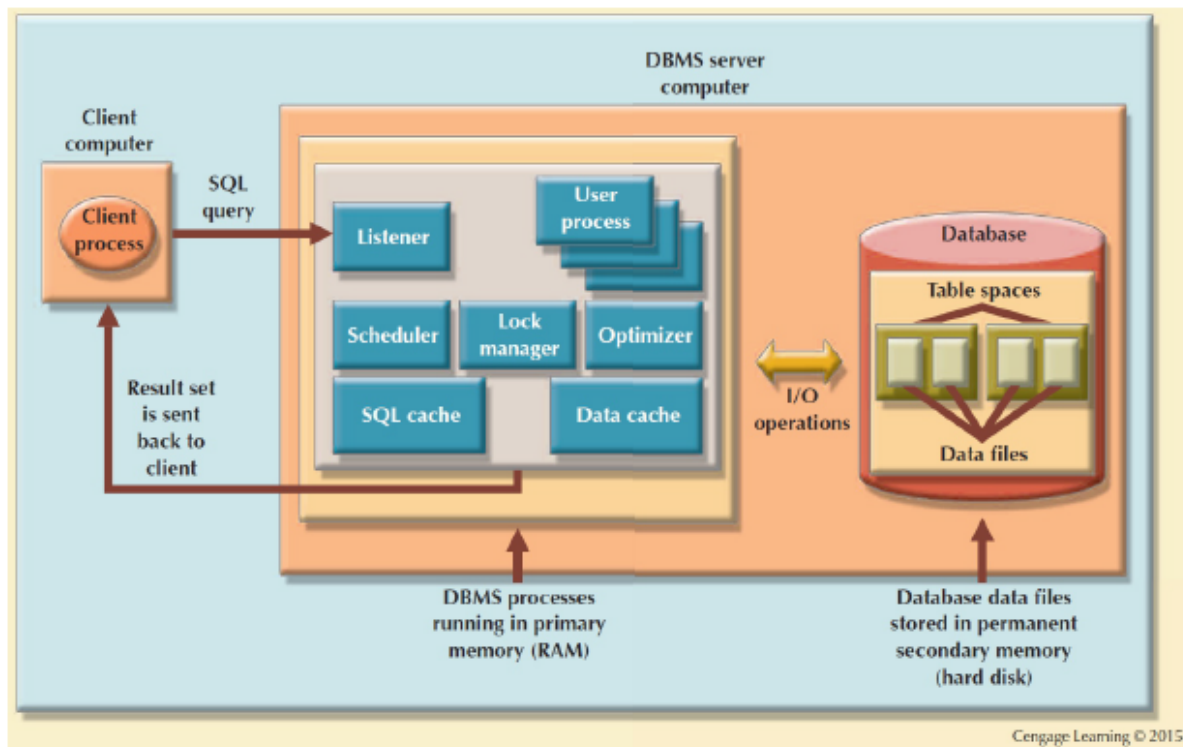
DBMS Architecture

- All data in a database are stored in **data files**
  - Data files automatically expand in predefined increments known as **extends**
- Data files are grouped in file groups or table spaces
  - **Table space** or **file group**: Logical grouping of several data files that store data with similar characteristics
- **Data cache** or **buffer cache**: Shared, reserved memory area
  - Stores most recently accessed data blocks in RAM

DBMS Architecture

- **SQL cache** or **procedure cache**: Stores most recently executed SQL statements or PL/SQL procedures

- DBMS retrieves data from permanent storage and places them in RAM. **Input/output request**: Low-level data access operation that reads or writes data to and from computer devices (note: reads fetch entire disk datablocks)

- Data cache is faster than working with data files

- Majority of performance-tuning activities focus on minimizing I/O operations

- *Listener*. The listener process listens for clients' requests and handles the processing of the SQL requests to other DBMS processes. Once a request is received, the listener passes the request to the appropriate user process.

- *User*. The DBMS creates a user process to manage each client session. Therefore, when you log on to the DBMS, you are assigned a user process. This process handles all requests you submit to the server. There are many user processes—at least one per logged-in client.

- *Scheduler*. The scheduler process organizes the concurrent execution of SQL requests. (See Chapter 10, Transaction Management and Concurrency Control.)

- *Lock manager*. This process manages all locks placed on database objects, including disk pages. (See Chapter 10.)

- *Optimizer*. The optimizer process analyzes SQL queries and finds the most efficient way to access the data. You will learn more about this process later in the chapter.

Database Query Optimization Modes

- Algorithms proposed for query optimization are based on:
  - Selection of the optimum order to achieve the fastest execution time
  - Selection of sites to be accessed to minimize communication costs
- Evaluated on the basis of:
  - Operation mode
  - Timing of its optimization
  - Type of information (used for optimization)

Classification of Operation Modes

- **Automatic query optimization**: DBMS finds the most cost-effective access path without user intervention

- **Manual query optimization**: Requires that the optimization be selected and scheduled by the end user or programmer

Based on Timing of Optimization

- **Static query optimization**: best optimization strategy is selected when the query is compiled by the DBMS
  - Takes place at compilation time
  - Best for embedded queries
- **Dynamic query optimization**: Access strategy is dynamically determined by the DBMS at run time, using the most up-to-date information about the database
  - Takes place at execution time; more processing o'head

Type of Information Used to Optimize

- **Statistically based query optimization algorithm**: Statistics are used by the DBMS to determine the best access strategy

- Statistical information is generated by DBMS through:

  - **Dynamic statistical generation mode – auto eval**

  - **Manual statistical generation mode - via user**

- **Rule-based query optimization algorithm**: based on a set of user-defined rules to determine the best query access strategy

Table 11.2 - Sample Database Statistics Measurements

DATABASE OBJECT	SAMPLE MEASUREMENTS
Tables	Number of rows, number of disk blocks used, row length, number of columns in each row, number of distinct values in each column, maximum value in each column, minimum value in each column, and columns that have indexes
Indexes	Number and name of columns in the index key, number of key values in the index, number of distinct key values in the index key, histogram of key values in an index, and number of disk pages used by the index
Environment Resources	Logical and physical disk block size, location and size of data files, and number of extends per data file

Cengage Learning © 2015

Commands to manually generate stats:
* COMPUTE STATISTICS (Oracle)
* ANALYZE TABLE (MySQL)
* UPDATE STATISTICS (SQL Server)
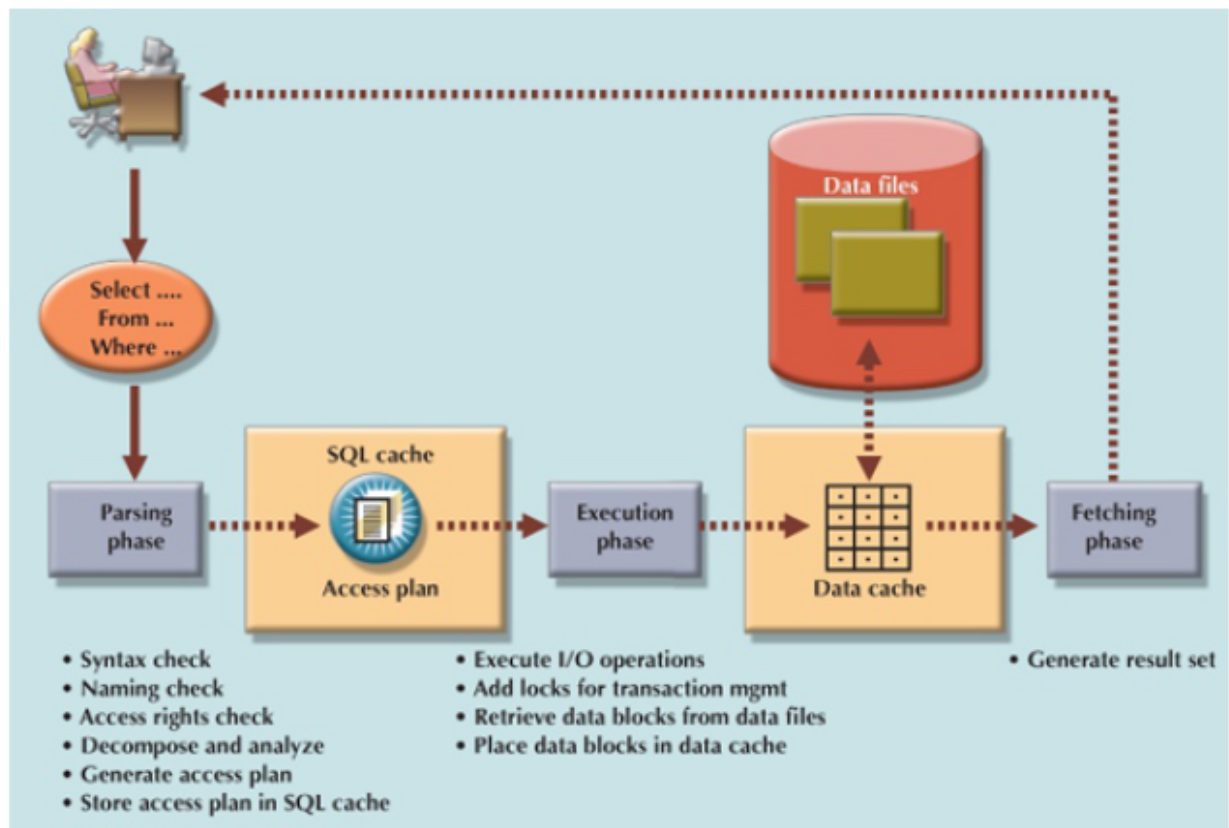
Query Processing

Parsing
- DBMS parses the SQL query and chooses the most efficient access/execution plan

Execution
- DBMS executes the SQL query using the chosen execution plan

Fetching
- DBMS fetches the data and sends the result set back to the client

SQL Parsing Phase (done by **query optimizer**)

- Query is broken down into smaller units

- Original SQL query is transformed into slightly different version of the original SQL code which is fully equivalent and more efficient

- **Query optimizer**: Analyzes SQL query and finds most efficient way to access data

- **Access plans**: DBMS-specific and translate client's SQL query into a series of complex I/O operations

SQL Parsing Phase

- If access plan already exists for query in SQL cache, DBMS reuses it

  - If not, optimizer evaluates various plans and chooses one to be placed in SQL cache for use

SQL Execution Phase

- All I/O operations indicated in the access plan are executed

  - Locks are acquired

  - Data are retrieved and placed in data cache

  - Transaction management commands are processed

Note that 'execution' here refers to executing the access plan, ie. fetching/sending data from/to the backend database – it does not refer to executing your SQL query. Query execution occurs in the next step (the 'fetching' step).

SQL Fetching Phase

- Rows of resulting query result set are returned to client

  - DBMS may use temporary table space to store temporary data

  - Database server coordinates the movement of the result set rows from the server cache to the client cache (eg will send blocks of rows to the client, wait for next request, send next block..)

Query Processing Bottlenecks

- Delay introduced in the processing of an I/O operation that slows the system
- Caused by the:
  - CPU – slow processor, rogue processes..
  - RAM – shared among running processes
  - Hard disk – disk speed, transfer rates..
  - Network – bandwidth shared among clients
  - Application code – bad user code, poor db design..

Indexes and Query Optimization

- Indexes
  - Help speed up data access
  - Facilitate searching, sorting, using aggregate functions, and join operations
  - Ordered set of values that contain the index key and pointers
  - More efficient than a full table scan

The reason for using an index is simple – provided we incur an upfront cost of creating (computing) one, runtime lookup costs using the index are vastly cheaper than doing full searches through non–indexed rows.

Eg. given this book, find all the places that discuss table updating..

A sample index

Indexes and Query Optimization

- **Data sparsity**: Number of different values a column could have; low sparsity => index might be useless

- Data structures used to implement indexes:
  - **Hash indexes**
  - **B-tree indexes**
  - **Bitmap indexes**

- DBMSs determine best type of index to use

Example of low sparsity: STU_SEX gender column in a table is either M or F, so indexing this is useless (we'd only have 2 keys, each with 1000s of records!). On the other hand, DOB is a column with relatively higher sparsity, and is worth creating an index for..

A hash index is based on an ordered list of hash values, computed from a key column, using a hashing algorithm - it is much faster to search through the hash values than search the columns. Each hash value then points to the actual (column) data.

A user query (eg. LNAME="Johnson") is converted to a hash 'key' which is then used to search through the pre-computed list of hash values and retrieve an exact match or a small set of values that are all stored

with the same key (as a result of 'hash collision').

B-tree index, bitmap index
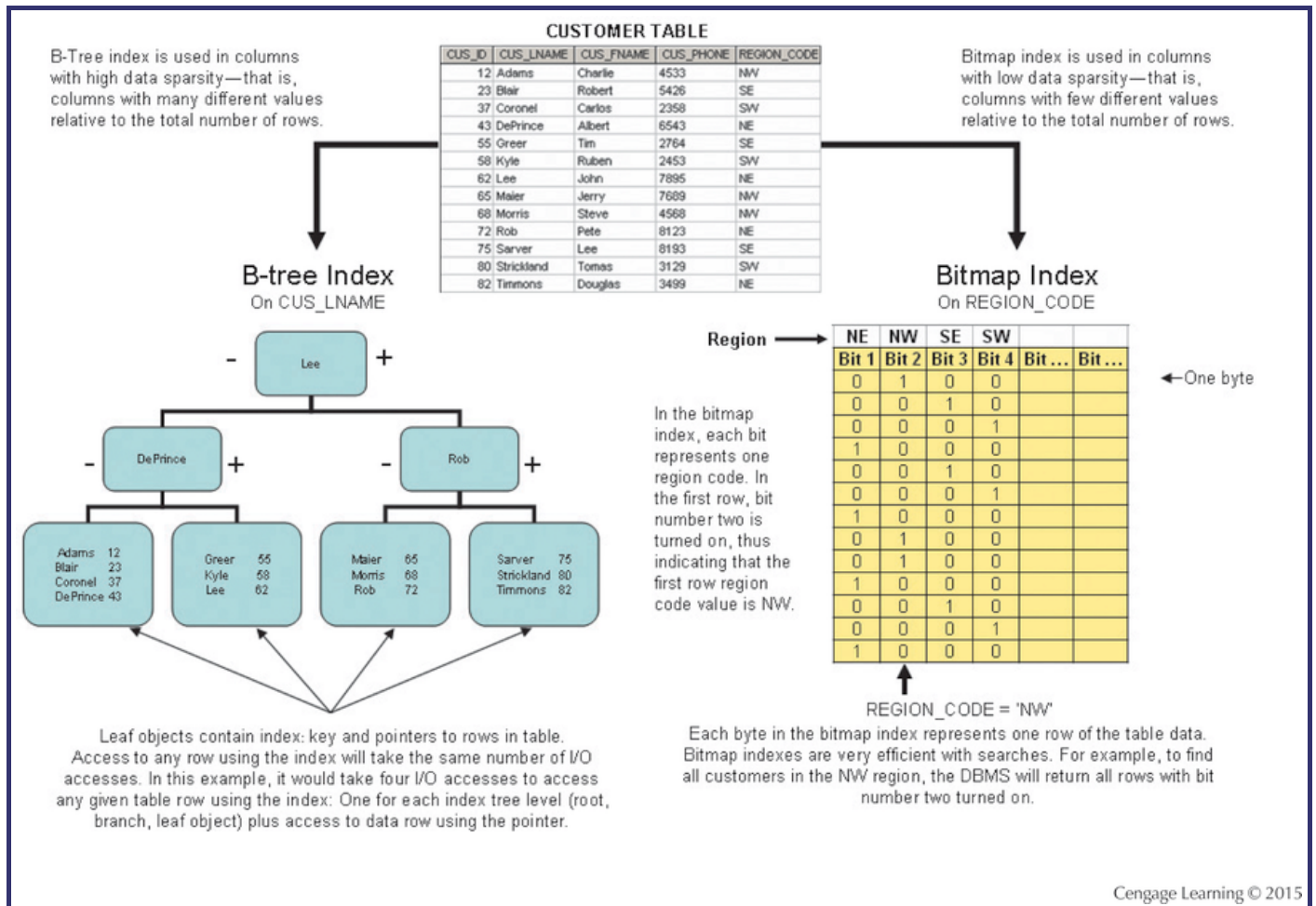
A bitmap index can use hardware arithmetic to look up actual rows in a table.

We use it when a column can have any of 'N' values, where N is small, eg. a customer's location in the US can only one of 6 values, like so:

In the above case, we'll create 6 bit columns (because there can be one of 6 values in the actual 'CustomerLocation' table column), eg like so: Alaska Hawaii Western Mountain Central Eastern

Then, for each customer, we set the bit to 1 where the customer lives, and the rest to 0. Further, we'll 'pad' with two more 0s, to use 8 bits (a byte). Eg three rows could be

```
0 0 1 0 0 0 0 0 (lives in CA)
0 1 0 0 0 0 0 0 (lives in Hawaii)
0 0 0 0 0 1 0 0 (lives in Maine)
...
```

So in the bitmap index, the number of rows will equal the # of rows in the actual table, and the number of columns, and the column names, will equal the possible values for the table column we are indexing.

The DBMS will store each above row as a byte. When we query for just the western region customers, the DBMS will '&' each row with a '00100000' "bitmask", ie efficiently go through each row in the bitmap index to filter just the 00100000 rows, for ex.

Index Selectivity

- Measure of the likelihood that an index will be used in query processing

- Indexes are used when a subset of rows from a large table is to be selected based on a given condition

- Index cannot always be used to improve performance

- **Function-based index**: Based on a specific SQL function or expression (eg. EMP_SALARY+EMP_COMMISSION)

Index selectivity: a measure of how likely an index will be used in a query. So we strive to create indexes that have high selectivity..

Indexes are useful when:
- an indexable column occurs in a WHERE or HAVING search expression
- an indexable column appears in a GROUP BY or ORDER BY clause
- MAX or MIN is applied to an indexable column
- there is high data sparsity on an indexable column

Worth creating indexes on single columns that appear in WHERE, HAVING, ORDER BY, GROUP BY and join conditions.

Rule-based vs cost-based (statistics-based)

Optimizer Choices

- **Rule-based optimizer**: Uses preset rules and points to determine the best approach to execute a query (choose a plan that minimizes the sum of rules' points (minimum cost))

- **Cost-based optimizer**: Uses algorithms based on statistics about objects being accessed (processing cost, RAM cost, I/O cost..) to determine the best approach to execute a query

Using Hints to Affect Optimizer Choices

- Optimizer might not choose the best execution plan
  - Makes decisions based on existing statistics, which might be old
  - Might choose less-efficient decisions
- **Optimizer hints**: Special instructions for the optimizer, embedded in the SQL command text

Table 11.5 - Optimizer Hints

HINT	USAGE
ALL_ROWS	Instructs the optimizer to minimize the overall execution time—that is, to minimize the time needed to return all rows in the query result set. This hint is generally used for batch mode processes. For example: SELECT /*+ ALL_ROWS */ * FROM PRODUCT WHERE P_QOH < 10;
FIRST_ROWS	Instructs the optimizer to minimize the time needed to process the first set of rows—that is, to minimize the time needed to return only the first set of rows in the query result set. This hint is generally used for interactive mode processes. For example: SELECT /*+ FIRST_ROWS */ * FROM PRODUCT WHERE P_QOH < 10;
INDEX(name)	Forces the optimizer to use the P_QOH_NDX index to process this query. For example: SELECT /*+ INDEX(P_QOH_NDX) */ * FROM PRODUCT WHERE P_QOH < 10;

Cengage Learning © 2015

SQL Performance Tuning

- Evaluated from client perspective
  - Most current relational DBMSs perform automatic query optimization at the server end
  - Most SQL performance optimization techniques are DBMS-specific and thus rarely portable
- Majority of performance problems are related to poorly written SQL code

Conditional Expressions

- Expressed within WHERE or HAVING clauses of a SQL statement
  - Restricts the output of a query to only rows matching conditional criteria
- Guidelines to write efficient conditional expressions in SQL code
  - Use simple columns or literals as operands
  - Numeric field comparisons are faster than character, date, and NULL comparisons

Conditional Expressions

- Equality comparisons are faster than inequality comparisons
- Transform conditional expressions to use literals
- Write equality conditions first when using multiple conditional expressions
- When using multiple AND conditions, write the condition most likely to be false first
- When using multiple OR conditions, put the condition most likely to be true first
- Avoid the use of NOT logical operator

Query Formulation

- Identify what columns and computations are required
- Identify source tables
- Determine how to join tables
- Determine what selection criteria are needed
- Determine the order in which to display the output

All of the above can be summarized like so: "know what you want, then find a good way to get it" [what do we want to return/compute/generate, how to best go about it (what SQL constructs to use, on what data)].

DBMS Performance Tuning

- Managing DBMS processes in primary memory and the structures in physical storage

- DBMS performance tuning at server end focuses on setting parameters used for:
  - Data cache
  - SQL cache
  - Sort cache
  - Optimizer mode

- **In-memory database**: Store large portions of the database in primary storage

DBMS Performance Tuning

- Recommendations for physical storage of databases:
  - Use **RAID** (Redundant Array of Independent Disks) to provide a balance between performance improvement and fault tolerance
  - Minimize disk contention
  - Put high-usage tables in their own table spaces
  - Assign separate data files in separate storage volumes for indexes, system, and high-usage tables

More info..

If you'd like more detail (MySQL-related, but the overall ideas are non-DB-specific), read this doc..