CMPG321 - Advanced Databases
Additional Notes
Semester 2

Last updated: 23 September 2021

Contents

Study Unit 1

Transaction Management and Concurrency Control

1.1 What Is a Transaction?

A **transaction** is a *logical* unit of work that must be entirely completed or entirely aborted; no intermediate states are acceptable.
A **consistent database state** is one in which all data integrity constraints are satisfied. To ensure consistency of the database, every transaction must begin with the database in a known consistent state.

1.1.1 Evaluating Transaction Results

Not all transactions update the database. The DBMS cannot guarantee that the semantic meaning of the transaction truly represents the real-world event.

1.1.2 Transaction Properties

Each individual transaction must display atomicity, consistency, isolation, and durability. These four properties are sometimes referred to as the ACID test.

- *Atomicity* requires that all operations (SQL requests) of a transaction be completed; if not, the transaction is aborted.
- *Consistency* indicates the permanence of the database's consistent state. A transaction takes a database from one consistent state to another. When a transaction is completed, the database must be in a consistent state. If any of the transaction parts violates an integrity constraint, the entire transaction is aborted.
- *Isolation* means that the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.
- *Durability* ensures that once transaction changes are done and committed, they cannot be undone or lost, even in the event of a system failure.

1.1.3 Transaction Management with SQL

- When a `COMMIT` statement is reached, all changes are permanently recorded within the database. The `COMMIT` statement automatically ends the SQL transaction.
- When a `ROLLBACK` statement is reached, all changes are aborted and the database is rolled back to its previous consistent state.
- The end of a program is successfully reached, in which case all changes are permanently recorded within the database. This action is equivalent to `COMMIT`.
- The program is abnormally terminated, in which case the database changes are aborted and the database is rolled back to its previous consistent state. This action is equivalent to `ROLLBACK`.

1.1.4 The Transaction Log

A DBMS uses a **transaction log** to keep track of all transactions that update the database. The DBMS uses the information stored in this log for a recovery requirement triggered by a `ROLLBACK` statement, a program's abnormal termination, or a system failure such as a network discrepancy or a disk crash. Some RDBMSs use the transaction log to recover a database forward to a currently consistent state.

While the DBMS executes transactions that modify the database, it also automatically updates the transaction log. The transaction log stores the following:

- A record for the beginning of the transaction.
- For each transaction component (SQL statement):
  - The type of operation being performed (`INSERT`, `UPDATE`, `DELETE`).
  - The names of the objects affected by the transaction (the name of the table).
  - The *"before"* and *"after"* values for the fields being updated.
  - Pointers to the previous and next transaction log entries for the same transaction.
- The ending (`COMMIT`) of the transaction.

1.2 Concurrency Control

Coordinating the simultaneous execution of transactions in a multiuser database system is known as **concurrency control**. The objective of concurrency control is to ensure the serializability of transactions in a multiuser database environment. To achieve this goal, most concurrency control techniques are oriented toward preserving the isolation property of concurrently executing transactions. Concurrency control is important because the simultaneous execution of transactions over a shared database can create several data integrity and consistency problems. The three main problems are lost updates, uncommitted data, and inconsistent retrievals.

1.2.1 Lost Updates

The **lost update** problem occurs when two concurrent transactions, T1 and T2, are updating the same data element and one of the updates is lost (overwritten by the other transaction).

1.2.2 Uncommitted Data

The phenomenon of **uncommitted data** occurs when two transactions, T1 and T2, are executed concurrently and the first transaction (T1) is rolled back after the second transaction (T2) has already accessed the uncommitted data—thus violating the isolation property of transactions.

1.2.3 Inconsistent Retrievals

Inconsistent retrievals occur when a transaction accesses data before and after one or more other transactions finish working with such data. For example, an inconsistent retrieval would occur if transaction T1 calculated some summary (aggregate) function over a set of data while another transaction (T2) was updating the same data. The problem is that the transaction might read some data before it is changed and other data after it is changed, thereby yielding inconsistent results.

1.2.4 The Scheduler

The **scheduler** is a special DBMS process that establishes the order in which the operations are executed within concurrent transactions. The scheduler *interleaves* the execution of database operations to ensure serializability and isolation of transactions. To determine the appropriate order, the scheduler bases its actions on concurrency control algorithms, such as locking or time stamping methods.

The scheduler's main job is to create a serializable schedule of a transaction's operations, in which the interleaved execution of the transactions (T1, T2, T3, etc.) yields the same results as if the transactions were executed in serial order (one after another).

The scheduler also makes sure that the computer's central processing unit (CPU) and storage systems are used efficiently.

1.3 Concurrency Control with Locking Methods

A **lock** guarantees exclusive use of a data item to a current transaction. In other words, transaction T2 does not have access to a data item that is currently being used by transaction T1. A transaction acquires a lock prior to data access; the lock is released (unlocked) when the transaction is completed.

The use of locks based on the assumption that conflict between transactions is likely is usually referred to as **pessimistic locking**.

Most multiuser DBMSs automatically initiate and enforce locking procedures. All lock information is handled by a **lock manager**, which is responsible for assigning and policing the locks used by the transactions.

1.3.1 Lock Granularity

Lock granularity indicates the level of lock use. Locking can take place at the following levels: database, table, page, row, or even field (attribute).

Database Level - In a **database-level lock**, the entire database is locked, thus preventing the use of any tables in the database by transaction T2 while transaction T1 is being executed. This level of locking is good for batch processes, but it is unsuitable for multiuser DBMSs.

Transactions T1 and T2 cannot access the same database concurrently *even when they use different tables.*

Table Level - In a **table-level lock**, the entire table is locked, preventing access to any row by transaction T2 while transaction T1 is using the table. If a transaction requires access to several tables, each table may be locked. However, two transactions can access the same database as long as they access different tables. Table-level locks, while less restrictive than database-level locks, cause traffic jams when many transactions are waiting to access the same table.

Page Level - In a **page-level lock**, the DBMS locks an entire **diskpage**. A diskpage, or page, is the equivalent of a *diskblock*, which can be described as a directly addressable section of a disk. A page has a fixed size, such as 4K, 8K, or 16K. For example, if you want to write only 73 bytes to a 4K page, the entire 4K page must be read from disk, updated in memory, and written back to disk. A table can span several pages, and a page can contain several rows of one or more tables. Page-level locks are currently the most frequently used locking method for multiuser DBMSs.

Row Level - A **row-level lock** is much less restrictive than the locks discussed earlier. The DBMS allows concurrent transactions to access different rows of the same table even when the rows are located on the same page. Although the row-level locking approach improves the availability of data, its management requires high overhead because a lock exists for each row in a table of the database involved in a conflicting transaction. Modern DBMSs automatically escalate a lock from a row level to a page level when the application session requests multiple locks on the same page.

Field Level - The **field-level lock** allows concurrent transactions to access the same row as long as they require the use of different fields (attributes) within that row. Although field-level locking clearly yields the most flexible multiuser data access, it is rarely implemented in a DBMS because it requires an extremely high level of computer overhead and because the row-level lock is much more useful in practice.

1.3.2 Lock Types

Binary - A **binary lock** has only two states: locked (1) or unlocked (0). If an object such as a database, table, page, or row is locked by a transaction, no other transaction can use that object. If an object is unlocked, any transaction can lock the object for its use. Every database operation requires that the affected object be locked. As a rule, a transaction must unlock the object after its termination. Therefore, every transaction requires a lock and unlock operation for each accessed data item. Such operations are automatically managed and scheduled by the DBMS.

Shared/Exclusive - An **exclusive lock** exists when access is reserved specifically for the transaction that locked the object. The exclusive lock must be used when the potential for conflict exists. A **shared lock** exists when concurrent transactions are granted read access on the basis of a common lock. A shared lock produces no conflict as long as all the concurrent transactions are read-only. Two transactions conflict only when at least one is a write transaction. Because the two read transactions can be safely executed at once, shared locks allow several read transactions to read the same data item concurrently. For example, if transaction T1 has a shared lock on data item X and transaction T2 wants to read data item X, T2 may also obtain a shared

lock on data item X. If transaction T2 updates data item X, an exclusive lock is required by T2 over data item X. The exclusive lock is granted if and only if no other locks are held on the data item (this condition is known as the **mutual exclusive rule**: only one transaction at a time can own an exclusive lock on an object.) Therefore, if a shared (or exclusive) lock is already held on data item X by transaction T1, an exclusive lock cannot be granted to transaction T2, and T2 must wait to begin until T1 commits. In other words, a shared lock will always block an exclusive (write) lock; hence, decreasing transaction concurrency.

Although the use of shared locks renders data access more efficient, a shared/exclusive lock schema increases the lock manager's overhead for several reasons:

- The type of lock held must be known before a lock can be granted.
- Three lock operations exist: `READ_LOCK` to check the type of lock, `WRITE_LOCK` to issue the lock, and `UNLOCK` to release the lock.
- The schema has been enhanced to allow a lock upgrade from shared to exclusive and a lock downgrade from exclusive to shared. Although locks prevent serious data inconsistencies, they can lead to two major problems:
  - The resulting transaction schedule might not be serializable.
  - The schedule might create deadlocks. A **deadlock** occurs when two transactions wait indefinitely for each other to unlock data. A database deadlock, which is similar to traffic gridlock in a big city, is caused when two or more transactions wait for each other to unlock data.

  Fortunately, both problems can be managed: serializability is attained through a locking protocol known as two-phase locking, and deadlocks can be managed by using deadlock detection and prevention techniques.

1.3.3 Two-Phase Locking to Ensure Serializability

Two-phase locking (2PL) defines how transactions acquire and relinquish locks. Two-phase locking guarantees serializability, but it does not prevent deadlocks. The two phases are:

1. A growing phase, in which a transaction acquires all required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.
2. A shrinking phase, in which a transaction releases all locks and cannot obtain a new lock.

The two-phase locking protocol is governed by the following rules:

- Two transactions cannot have conflicting locks.
- No unlock operation can precede a lock operation in the same transaction.
- No data is affected until all locks are obtained—that is, until the transaction is in its locked point.

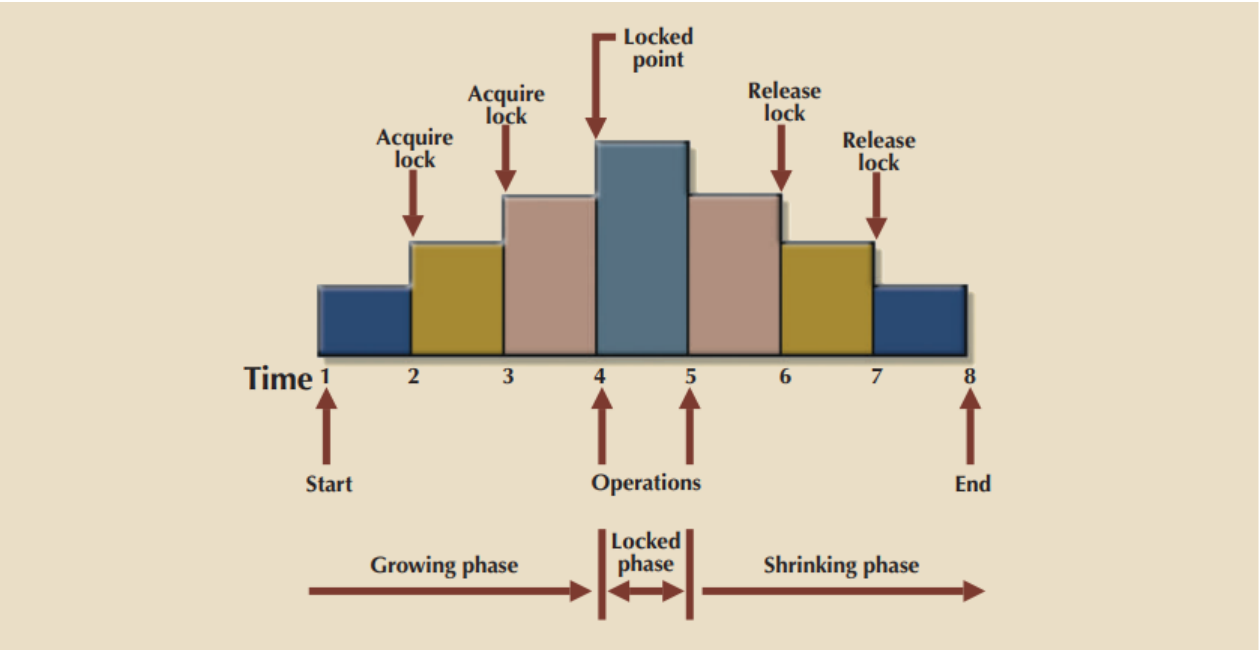

Figure 1.1: Two-phase locking process illustrated

1.3.4 Deadlocks

A deadlock occurs when two transactions wait indefinitely for each other to unlock data. For example, a deadlock occurs when two transactions, T1 and T2, exist in the following mode:

T1 = access data items X and Y
T2 = access data items Y and X

If T1 has not unlocked data item Y, T2 cannot begin; if T2 has not unlocked data item X, T1 cannot continue. Consequently, T1 and T2 each wait for the other to unlock the required data item. Such a deadlock is also known as a **deadly embrace**. Note that deadlocks are possible only when one of the transactions wants to obtain an exclusive lock on a data item; no deadlock condition can exist among *shared* locks.

The three basic techniques to control deadlocks are:

- *Deadlock prevention.* A transaction requesting a new lock is aborted when there is the possibility that a deadlock can occur. If the transaction is aborted, all changes made by this transaction are rolled back and all locks obtained by the transaction are released. The transaction is then rescheduled for execution. Deadlock prevention works because it avoids the conditions that lead to deadlocking.

- *Deadlock detection.* The DBMS periodically tests the database for deadlocks. If a deadlock is found, the "victim" transaction is aborted (rolled back and restarted) and the other transaction continues.

- *Deadlock avoidance.* The transaction must obtain all of the locks it needs before it can be executed. This technique avoids the rolling back of conflicting transactions by requiring that locks be obtained in succession. However, the serial lock assignment required in deadlock avoidance increases action response times.

1.4 Concurrency Control with Time Stamping Methods

The **time stamping** approach to scheduling concurrent transactions assigns a global, unique time stamp to each transaction. The time stamp value produces an explicit order in which transactions are submitted to the DBMS. Time stamps must have two properties: uniqueness and monotonicity. **Uniqueness** ensures that no equal time stamp values can exist, and **monotonicity**[1] ensures that time stamp values always increase.

1.4.1 Wait/Die and Wound/Wait Schemes

Using the wait/die scheme:

- If the transaction requesting the lock is the older of the two transactions, it will wait until the other transaction is completed and the locks are released.
- If the transaction requesting the lock is the younger of the two transactions, it will die (roll back) and is rescheduled using the same time stamp.

In short, in the **wait/die** scheme, the older transaction waits for the younger one to complete and release its locks. In the wound/wait scheme:

- If the transaction requesting the lock is the older of the two transactions, it will *preempt (wound)* the younger transaction by rolling it back. T1 preempts T2 when T1 rolls back T2. The younger, preempted transaction is rescheduled using the same time stamp.
- If the transaction requesting the lock is the younger of the two transactions, it will wait until the other transaction is completed and the locks are released.

In short, in the **wound/wait** scheme, the older transaction rolls back the younger transaction and reschedules it. In both schemes, one of the transactions waits for the other transaction to finish and release the locks.

1.5 Concurrency Control with Optimistic Methods

The **optimistic approach** is based on the assumption that the majority of database operations do not conflict. The optimistic approach requires neither locking nor time stamping techniques. Instead, a transaction is executed without restrictions until it is committed. Using an optimistic approach, each transaction moves through two or three phases, referred to as *read*, *validation*, and *write*.

- During the *read phase*, the transaction reads the database, executes the needed computations, and makes the updates to a private copy of the database values. All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions.
- During the *validation phase*, the transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database. If the validation test is positive, the transaction goes to the write phase. If the validation test is negative, the transaction is restarted and the changes are discarded.
- During the *write phase*, the changes are permanently applied to the database.

[1] The term monotonicity is part of the standard concurrency control vocabulary.

1.6 Database Recovery Management

Database recovery restores a database from a given state (usually inconsistent) to a previously consistent state. Recovery techniques are based on the **atomic transaction property**: all portions of the transaction must be treated as a single, logical unit of work in which all operations are applied and completed to produce a consistent database.

Critical events can cause a database to stop working and compromise the integrity of the data. Examples of critical events are:

- *Hardware/software failures.* A failure of this type could be a hard disk media failure, a bad capacitor on a motherboard, or a failing memory bank. Other causes of errors under this category include application program or operating system errors that cause data to be overwritten, deleted, or lost.
- *Human-caused incidents.* This type of event can be categorized as unintentional or intentional.
  - An unintentional failure is caused by a careless end user. Such errors include deleting the wrong rows from a table, pressing the wrong key on the keyboard, or shutting down the main database server by accident.
  - Intentional events are of a more severe nature and normally indicate that the company data is at serious risk. Under this category are security threats caused by hackers trying to gain unauthorized access to data resources and virus attacks caused by disgruntled employees trying to compromise the database operation and damage the company.
- *Natural disasters.* This category includes fires, earthquakes, floods, and power failures.

1.6.1 Transaction Recovery

Database transaction recovery uses data in the transaction log to recover a database from an inconsistent state to a consistent state.There are four important concepts that affect the recovery process:

- The **write-ahead-log protocol** ensures that transaction logs are always written before any database data is actually updated.
- **Redundant transaction logs** ensure that a physical disk failure will not impair the DBMS's ability to recover data.
- Database **buffers** are temporary storage areas in primary memory used to speed up disk operations. To improve processing time, the DBMS software reads the data from the physical disk and stores a copy of it on a "buffer" in primary memory.
- Database **checkpoints** are operations in which the DBMS writes all of its updated buffers in memory (also known as *dirty buffers*) to disk. While this is happening, the DBMS does not execute any other requests. A checkpoint operation is also registered in the transaction log. As a result of this operation, the physical database and the transaction log will be in sync.

The database recovery process involves bringing the database to a consistent state after a failure. Transaction recovery procedures generally make use of deferred-write and write-through techniques.

When the recovery procedure uses a **deferred-write technique** (also called a **deferred update**), the transaction operations do not immediately update the physical database. Instead, only the transaction log is updated. The database is physically updated only with data from committed transactions, using information from the transaction log. If the transaction aborts before it reaches its commit point, no changes (no `ROLLBACK` or undo) need to be made to the database because it was never updated. The recovery process for all started and committed transactions (before the failure) follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data is already saved.

3. For a transaction that performed a commit operation after the last checkpoint, the DBMS uses the transaction log records to redo the transaction and update the database, using the "after" values in the transaction log. The changes are made in ascending order, from oldest to newest.

4. For any transaction that had a `ROLLBACK` operation after the last checkpoint or that was left active (with neither a `COMMIT` nor a `ROLLBACK`) before the failure occurred, nothing needs to be done because the database was never updated.

When the recovery procedure uses a **write-through technique** (also called an **immediate update**), the database is immediately updated by transaction operations during the transaction's execution, even before the transaction reaches its commit point. If the transaction aborts before it reaches its commit point, a `ROLLBACK` or undo operation needs to be done to restore the database to a consistent state. In that case, the `ROLLBACK` operation will use the transaction log "before" values. The recovery process follows these steps:

1. Identify the last checkpoint in the transaction log. This is the last time transaction data was physically saved to disk.

2. For a transaction that started and was committed before the last checkpoint, nothing needs to be done because the data is already saved.

3. For a transaction that was committed after the last checkpoint, the DBMS re-does the transaction, using the "after" values of the transaction log. Changes are applied in ascending order, from oldest to newest.

4. For any transaction that had a `ROLLBACK` operation after the last checkpoint or that was left active (with neither a `COMMIT` nor a `ROLLBACK`) before the failure occurred, the DBMS uses the transaction log records to `ROLLBACK` or undo the operations, using the "before" values in the transaction log. Changes are applied in reverse order, from newest to oldest.

Study Unit 2

Database Performance Tuning and Query Optimization

2.1 Database Performance-Tuning Concepts

One of the main functions of a database system is to provide timely answers to end users. End users interact with the DBMS through the use of queries to generate information, using the following sequence:

1. The end-user (client-end) application generates a query.
2. The query is sent to the DBMS (server end).
3. The DBMS (server end) executes the query.
4. The DBMS sends the resulting data set to the end-user (client-end) application.

Database performance tuning refers to a set of activities and procedures designed to reduce the response time of the database system-that is, to ensure that an end-user query is processed by the DBMS in the minimum amount of time.

Good database performance starts with good database design. *No amount of fine tuning will make a poorly designed database perform as well as a well-designed database.*

2.1.1 Performance Tuning: Client and Server

In general, database performance-tuning activities can be divided into those on the client side and those on the server side.

On the client side, the objective is to generate a SQL query that returns the correct answer in the least amount of time, using the minimum amount of resources at the server end. The activities required to achieve that goal are commonly referred to as **SQL performance tuning**.

On the server side, the DBMS environment must be properly configured to respond to clients' requests in the fastest way possible, while making optimum use of existing resources. The activities required to achieve that goal are commonly referred to as **DBMS performance tuning**.

2.1.2 DBMS Architecture

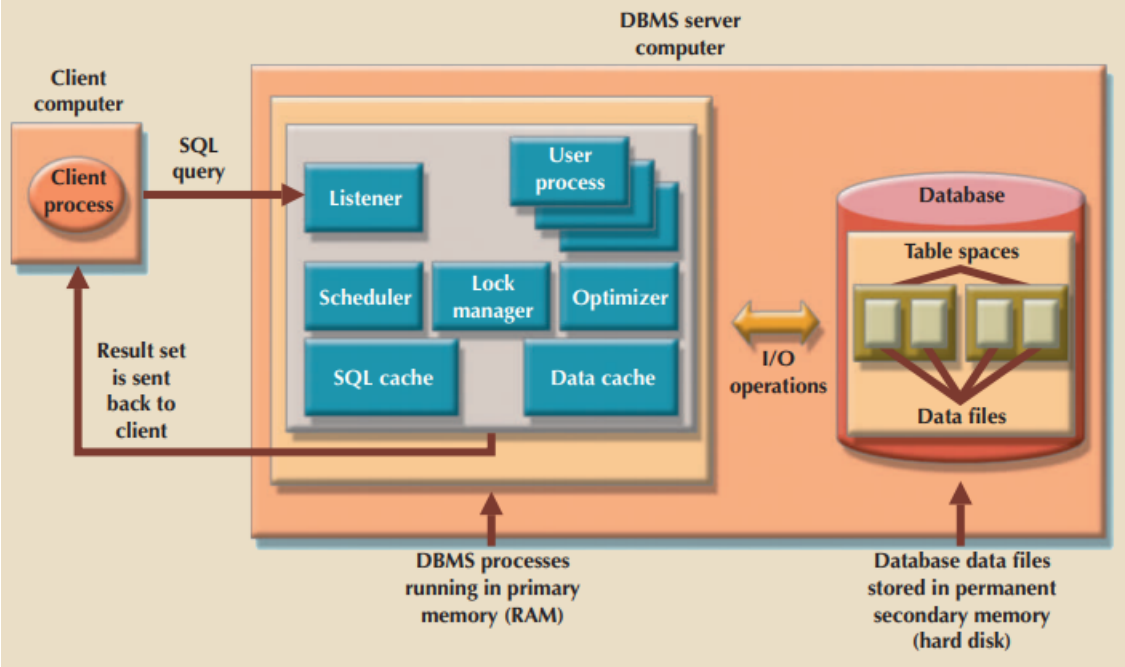

Figure 2.1: DBMS Architecture

All data in a database is stored in **data files**. A data file can contain rows from a single table, or it can contain rows from many different tables. Data files can automatically expand as required in predefined increments known as **extents**.

Data files are generally grouped in file groups or table spaces. A **table space** or **file group** is a logical grouping of several data files that store data with similar characteristics.

The **data cache**, or **buffer cache**, is a shared, reserved memory area that stores the most recently accessed data blocks in RAM.

The **SQL cache**, or **procedure cache**, is a shared, reserved memory area that stores the most recently executed SQL statements or PL/SQL procedures, including triggers and functions.

To move data from permanent storage (data files) to RAM (data cache), the DBMS issues I/O requests and waits for the replies. An **input/output (I/O) request** is a lowlevel data access operation that reads or writes data to and from computer devices.

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.

Lock manager - This process manages all locks placed on database objects, including disk pages.

Optimizer - The optimizer process analyzes SQL queries and finds the most efficient way to access the data.

2.1.3 Database Query Optimization Modes

Most of the algorithms proposed for query optimization are based on two principles:

- The selection of the optimum execution order to achieve the fastest execution time
- The selection of sites to be accessed to minimize communication costs

Within those two principles, a query optimization algorithm can be evaluated on the basis of its *operation mode* or the *timing of its optimization*. Operation modes can be classified as manual or automatic.

Automatic query optimization means that the 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.

Query optimization algorithms can also be classified according to when the optimization is done. Within this timing classification, query optimization algorithms can be static or dynamic.

Static query optimization takes place at compilation time. In other words, the best optimization strategy is selected when the query is compiled by the DBMS.

Dynamic query optimization takes place at execution time. Database access strategy is defined when the program is executed.

Finally, query optimization techniques can be classified according to the type of information that is used to optimize the query. For example, queries may be based on statistically based or rule-based algorithms.

A **statistically based query optimization algorithm** uses statistical information about the database. The statistical information is managed by the DBMS and is generated in one of two different modes: dynamic or manual. In the **dynamic statistical generation mode**, the DBMS automatically evaluates and updates the statistics after each data access operation. In the **manual statistical generation mode**, the statistics must be updated periodically through a user-selected utility.

A **rule-based query optimization algorithm** is based on a set of user-defined rules to determine the best query access strategy.

2.1.4 Database Statistics

Another DBMS process that plays an important role in query optimization is gathering database statistics. The term **database statistics** refers to a number of measurements about database objects.

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

2.2 Query Processing

In simple terms, the DBMS processes a query in three phases:

1. *Parsing* - The DBMS parses the SQL query and chooses the most efficient access/ execution plan.
2. *Execution* - The DBMS executes the SQL query using the chosen execution plan.
3. *Fetching* - The DBMS fetches the data and sends the result set back to the client.

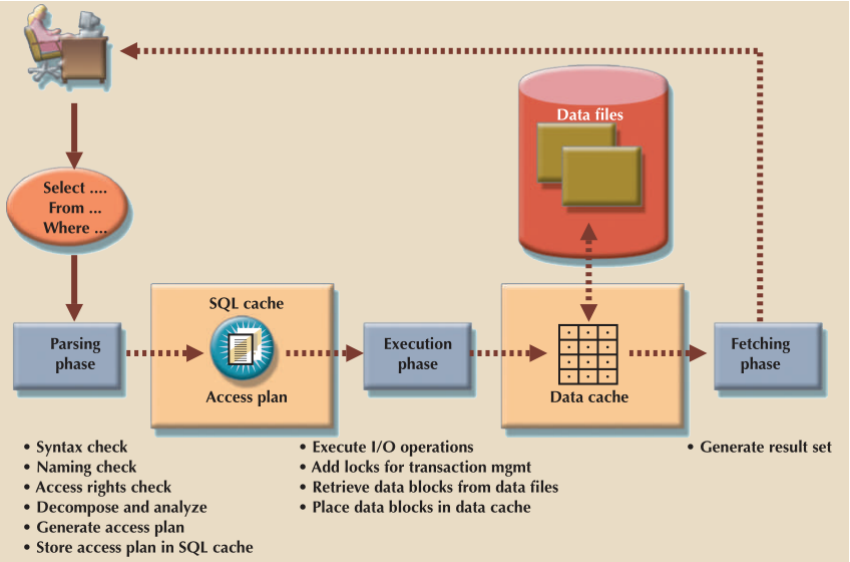

Figure 2.2: Query Processing

2.2.1 SQL Parsing Phase

The SQL parsing activities are performed by the **query optimizer**, which analyzes the SQL query and finds the most efficient way to access the data. This process is the most time-consuming phase in query processing. Parsing a SQL query requires several steps, in which the SQL query is:

- Validated for syntax compliance
- Validated against the data dictionary to ensure that table names and column names are correct
- Validated against the data dictionary to ensure that the user has proper access rights
- Analysed and decomposed into more atomic components
- Optimized through transformation into a fully equivalent but more efficient SQL query
- Prepared for execution by determining the most efficient execution or access plan

OPERATION	DESCRIPTION
Table scan (full)	Reads the entire table sequentially, from the first row to the last, one row at a time (slowest)
Table access (row ID)	Reads a table row directly, using the row ID value (fastest)
Index scan (range)	Reads the index first to obtain the row IDs and then accesses the table rows directly (faster than a full table scan)
Index access (unique)	Used when a table has a unique index in a column
Nested loop	Reads and compares a set of values to another set of values, using a nested loop style (slow)
Merge	Merges two data sets (slow)
Sort	Sorts a data set (slow)

Figure 2.3: Sample DBMS Access Plan I/O Operations

2.2.2 SQL Execution Phase

In this phase, all I/O operations indicated in the access plan are executed. When the execution plan is run, the proper locks-if needed-are acquired for the data to be accessed, and the data is retrieved from the data files and placed in the DBMS's data cache.

2.2.3 SQL Fetching Phase

After the parsing and execution phases are completed, all rows that match the specified condition(s) are retrieved, sorted, grouped, and aggregated (if required). During the fetching phase, the rows of the resulting query result set are returned to the client. The DBMS might use temporary table space to store temporary data.

2.2.4 Query Processing Bottlenecks

A **query processing bottleneck** is a delay introduced in the processing of an I/O operation that causes the overall system to slow down. Within a DBMS, five components typically cause bottlenecks:

- *CPU* - The CPU processing power of the DBMS should match the system's expected work load. A high CPU utilization might indicate that the processor speed is too slow for the amount of work performed.
- *RAM* - The DBMS allocates memory for specific usage, such as data cache and SQL cache. RAM must be shared among all running processes, including the operating system and DBMS.
- *Hard disk* - Other common causes of bottlenecks are hard disk speed and data transfer rates.
- *Network* - In a database environment, the database server and the clients are connected via a network. All networks have a limited amount of bandwidth that is shared among all clients.
- *Application code* - Not all bottlenecks are caused by limited hardware resources. Two of the most common sources of bottlenecks are inferior application code and poorly designed databases.

2.3 Indexes and Query Optimization

Indexes are crucial in speeding up data access because they facilitate searching, sorting, and using aggregate functions and even join operations. The improvement in data access speed occurs because an index is an ordered set of values that contains the index key and pointers. If there is no index, the DBMS will perform a full-table scan. One measure that determines the need for an index is the data sparsity of the column you want to index. **Data sparsity** refers to the number of different values a column could have.

Most DBMSs implement indexes using one of the following data structures:

- **Hash index** - A hash index is based on an ordered list of hash values. A hash algorithm is used to create a hash value from a key column. This value points to an entry in a hash table, which in turn points to the actual location of the data row. This type of index is good for simple and fast lookup operations based on equality conditions.
- **B-tree index** - The B-tree index is an ordered data structure organized as an upside-down tree. The index tree is stored separately from the data. The lower-level leaves of the B-tree index contain the pointers to the actual data rows. B-tree indexes are "self-balanced," which means that it takes approximately the same amount of time to access any given row in the index.
- **Bitmap index** - A bitmap index uses a bit array (0s and 1s) to represent the existence of a value or condition. These indexes are used mostly in data warehouse applications in tables with a large number of rows in which a small number of column values repeat many times. Bitmap indexes tend to use less space than B-tree indexes because they use bits instead of bytes to store their data.

2.4 Optimizer Choices

Each DBMS has its own algorithms for deter- mining the most efficient way to access the data. The query optimizer can operate in one of two modes:

- A **rule-based optimizer** uses preset rules and points to determine the best approach to execute a query. The rules assign a "fixed cost" to each SQL operation; the costs are then added to yield the cost of the execution plan.
- A **cost-based optimizer** uses sophisticated algorithms based on statistics about the objects being accessed to determine the best approach to execute a query. In this case, the optimizer process adds up the processing cost, the I/O costs, and the resource costs to determine the total cost of a given execution plan.

2.4.1 Using Hints to Affect Optimizer Choices

Sometimes the end user would like to change the optimizer mode for the current SQL statement. To do that, you need to use hints. **Optimizer hints** are special instructions for the optimizer that are embedded inside the SQL command text.

2.5 SQL Performance Tuning

SQL performance tuning is evaluated from the client perspective. Therefore, the goal is to illustrate some common practices used to write efficient SQL code. A few words of caution are

appropriate:

- Most current-generation relational DBMSs perform automatic query optimization at the server end.
- Most SQL performance optimization techniques are DBMS-specific and, therefore, are rarely portable, even across different versions of the same DBMS. Part of the reason for this behaviour is the constant advancement in database technologies.

2.5.1 Index Selectivity

Indexes are the most important technique used in SQL performance optimization. The key is to know when an index is used. As a general rule, indexes are likely to be used:

- When an indexed column appears by itself in the search criteria of a `WHERE` or `HAVING` clause.
- When an indexed column appears by itself in a `GROUP BY` or `ORDER BY` clause
- When a `MAX` or `MIN` function is applied to an indexed column
- When the data sparsity on the indexed column is high

The objective is to create indexes with high selectivity. **Index selectivity** is a measure of the likelihood that an index will be used in query processing. Here are some general guidelines for creating and using indexes:

- Create indexes for each single attribute used in a `WHERE`, `HAVING`, `ORDER BY`, or `GROUP BY` clause.
- Do not use indexes in small tables or tables with low sparsity.
- Declare primary and foreign keys so the optimizer can use the indexes in join operations.
- Declare indexes in join columns other than PK or FK.

You cannot always use an index to improve performance. The reason is that in some DBMSs, indexes are ignored when you use functions in the table attributes. However, major databases such as Oracle, SQL Server, and DB2 now support function-based indexes. A **function-based index** is an index based on a specific SQL function or expression. Function-based indexes are especially useful when dealing with derived attributes.

2.5.2 Conditional Expressions

A conditional expression is normally placed within the `WHERE` or `HAVING` clauses of a SQL statement. Also known as conditional criteria, a conditional expression restricts the output of a query to only the rows that match the conditional criteria. Most of the query optimization techniques mentioned below are designed to make the optimizer's work easier. The following common practices are used to write efficient conditional expressions in SQL code.

- Use simple columns or literals as operands in a conditional expression-avoid the use of conditional expressions with functions whenever possible. Comparing the contents of a single column to a literal is faster than comparing to expressions.

- Numeric field comparisons are faster than character, date, and `NULL` comparisons. In search conditions, comparing a numeric attribute to a numeric literal is faster than comparing a character attribute to a character literal.
- Equality comparisons are generally faster than inequality comparisons.
- Whenever possible, transform conditional expressions to use literals.
- When using multiple conditional expressions, write the equality conditions first.
- If you use 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.
- Whenever possible, try to avoid the use of the `NOT` logical operator.

2.6 Query formulation

Queries are usually written to answer questions. This section focuses on `SELECT` queries because they are the queries you will find in most applications. To formulate a query, you would normally follow these steps:

1. Identify what columns and computations are required.
    - Do you need simple expressions?
    - Do you need aggregate functions?
    - Determine the granularity of the raw data required for your output.
2. Identify the source tables.
3. Determine how to join the tables.
4. Determine what selection criteria are needed.
    - Simple comparisons
    - Single value to multiple values
    - Nested comparisons
    - Grouped data selection
5. Determine the order in which to display the output

2.7 DBMS Performance Tuning

DBMS performance tuning includes global tasks such as managing the DBMS processes in primary memory and managing the structures in physical storage.

DBMS performance tuning at the server end focuses on setting the parameters used for:

- Data cache
- SQL cache
- Sort cache
- Optimizer mode

From the performance point of view, it would be optimal to have the entire database stored in primary memory to minimize costly disk access. This is why several database vendors offer in-memory database options for their main products. **In-memory database** systems are optimized to store large portions (if not all) of the database in primary (RAM) storage rather than secondary (disk) storage.

Note the following general recommendations for physical storage of databases:

- Use **I/O accelerators** - This type of device uses flash solid state drives (SSDs) to store the database.
- Use **RAID** (Redundant Array of Independent Disks) to provide both performance improvement and fault tolerance, and a balance between them.
- Minimize disk contention. A database should have at least the following table spaces:
  - *System table space* - This is used to store the data dictionary tables.
  - *User data table space* - This is used to store end-user data.
  - *Index table space* - This is used to store indexes.
  - *Temporary table space* - This is used as a temporary storage area for merge, sort, or set aggregate operations.
  - *Rollback segment table space* - This is used for transaction-recovery purposes.
- Put high-usage tables in their own table spaces so the database minimizes conflict with other tables.
- Assign separate data files in separate storage volumes for the indexes, system, and high-usage tables.
- Take advantage of the various table storage organizations available in the database. For example, in Oracle consider the use of index-organized tables (IOT); in SQL Server, consider clustered index tables. An **index-organized table** (or **clustered index table**) is a table that stores the end-user data and the index data in consecutive locations on permanent storage.
- Partition tables based on usage.
- Use denormalized tables where appropriate.
- Store computed and aggregate attributes in tables.

Study Unit 3

Distributed Database Management Systems

Study Unit 4

SQL Fundamentals 1

Study Unit 5

SQL Fundamentals 2

Study Unit 6

Business Intelligence and Data Warehouses

Study Unit 7

Big Data and NoSQL

Study Unit 8

Database Administration and Security