Concurrency Control

Many DBMSs allow users to undertake simultaneous operations on the database. If these operations are not controlled, the accesses may interfere with one another and the database can become inconsistent.
To overcome this, the DBMS implements a **concurrency control** protocol that prevents database accesses from interfering with one another.
The coordination of the simultaneous execution of transactions in a multiuser database system is known as **concurrency control**. It is the process of managing simultaneous operations on the database **control** without having them interfere with one another.
The objective of concurrency control is to ensure the serializability of transactions in a multiuser database environment.
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.

Lost Updates/Multiple update problem
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).
An apparently successfully completed update operation by one user can be overridden by another user.

Uncommitted Data/dirty reads/uncommitted dependency problem

One transaction is allowed to see the intermediate results of another transaction before it has committed
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.

Inconsistent Retrievals/Incorrect analysis problem
When a transaction reads several values from the database but a second transaction updates some of them during the execution of the first.
Inconsistent retrievals occur when a transaction accesses data before and after another transaction(s) finish working with such data.
Another problem can occur when a transaction T rereads a data item it has previously read but, in between, another transaction has modified it. Thus, T receives two different values for the same data item. This is sometimes referred to as a **nonrepeatable** (or **fuzzy**) **read**.
A similar problem can occur if transaction T executes a query that retrieves a set of tuples from a relation satisfying a certain predicate, re-executes the query at a later time

but finds that the retrieved set contains an additional (**phantom**) tuple that has been inserted by another transaction in the meantime. This is sometimes referred to as a **phantom read**.

The Scheduler

During multiple transaction execution the following can arise:

- Severe problems can arise when two or more concurrent transactions are executed.
- A database transaction involves a series of database I/O operations that take the database from one consistent state to another.
- That database consistency can be ensured only before and after the execution of transactions.
- A database always moves through an unavoidable temporary state of inconsistency during a transaction's execution if such transaction updates multiple tables/rows. (If the transaction contains only one update, then there is no temporary inconsistency.)

That temporary inconsistency exists because a computer executes the operations serially, one after another. During this serial process, the isolation property of transactions prevents them from accessing the data not yet released by other transactions. The job of the scheduler is even more important today, with the use of multicore processors that have the capability of executing several instructions at the same time. What would happen if two transactions executed concurrently and they were accessing the same data?

The **scheduler** is a special DBMS process that:

a) Establishes the order in which the operations within concurrent transactions are executed. The scheduler *interleaves* the execution of database operations to ensure serializability and isolation of transactions.

- A schedule is a sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions. A transaction comprises a sequence of operations consisting of read and/or write actions to the database, followed by a commit or abort action.
- To determine the appropriate order, the scheduler bases its actions on concurrency control algorithms, such as locking or time stamping methods,
- However, it is important to understand that not all transactions are serializable. The DBMS determines what transactions are serializable and proceeds to interleave the execution of the transaction's operations.
- Transactions that are not serializable are executed on a first-come, first-served basis by the DBMS. The scheduler's main job is to create a serializable schedule of a transaction's operations.
- A **serializable schedule** is a 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).

- In a serial schedule, the transactions are performed in serial order. For example, if we have two transactions T1 and T2, serial order would be T1 followed by T2, or T2 followed by T1. Thus, in serial execution there is no interference between transactions, since only one is executing at any given time. However, there is no guarantee that the results of all serial executions of a given set of transactions will be identical. In banking, for example, it matters whether interest is calculated on an account before a large deposit is made or after.

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

- If there were no way to schedule the execution of transactions, all transactions would be executed on a first-come, first-served basis. The problem with that approach is that processing time is wasted when the CPU waits for a READ or WRITE operation to finish, thereby losing several CPU cycles. In short, first-come, first-served scheduling tends to yield unacceptable response times within the multiuser DBMS environment. Therefore, some other scheduling method is needed to improve the efficiency of the overall system.

c) The scheduler facilitates data isolation to ensure that two transactions do not update the same data element at the same time. Database operations might require READ and/or WRITE actions that produce conflicts

CONCURRENCY CONTROL WITH LOCKING METHODS

A **lock** guarantees exclusive use of a data item to a current transaction.

- For instance if 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 so that another transaction can lock the data item for its exclusive use.

This series of locking actions assumes that there is a likelihood of concurrent transactions attempting to manipulate the same data item at the same time. The use of locks based on the assumption that conflict between transactions is likely to occur is often referred to as **pessimistic locking**.

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

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).

- 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 Tl is being executed.
- 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.

- In a **page-level lock**, the DBMS will lock 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 **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.
- 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.

Lock Types

Binary Lock

- A **binary lock** has only two states: locked (1) or unlocked (0). If an object—that is, 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
- Note that the lock and unlock features eliminate the lost update problem because the lock is not released until the WRITE statement is completed.

Shared/Exclusive Locks

- The labels "shared" and "exclusive" indicate the nature of the lock. 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.
- A **shared lock** is issued when a transaction wants to read data from the database and no exclusive lock is held on that data item.
- An **exclusive lock** is issued when a transaction wants to update (write) a data item and no locks are currently held on that data item by any other transaction.
- Using the shared/exclusive locking concept, a lock can have three states: unlocked, shared (read), and exclusive (write).

Two-Phase Locking to Ensure Serializability

Two-phase locking 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 any 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 are affected until all locks are obtained—that is, until the transaction is in its locked point.

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**.

In a real-world DBMS, many more transactions can be executed simultaneously, thereby increasing the probability of generating deadlocks. 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, one of the transactions (the "victim") 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.

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,

- **Monotonicity** ensures that time stamp values always increase.

All database operations (Read and Write) within the same transaction must have the same time stamp.
The DBMS executes conflicting operations in time stamp order, thereby ensuring serializability of the transactions.
If two transactions conflict, one is stopped, rolled back, rescheduled, and assigned a new time stamp value.
Time stamping methods are used to manage concurrent transaction execution. It is you uses two schemes to decide which transaction is rolled back and which continues executing:

- The wait/die scheme
- the wound/wait scheme

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.

N/B In short, in the **wait/die** scheme, the older transaction waits for the younger 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.

N/B 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.

CONCURRENCY CONTROL WITH OPTIMISTIC METHODS

The optimistic approach is based on the assumption that the majority of the 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.

The optimistic approach is acceptable for most read or query database systems that require few update transactions.