CONCURRENCY CONTROL

The coordination of 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. 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.
- Inconsistent retrievals.

The lost update

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).

Assume that two concurrent transactions occur T1 and T2 that update say prd-qoh value for some item in the product table. These transactions are:

Transaction	Computation
T1: Purchase 100 items	PROD-QOH=PROD-QOH+100
T2: Sell 30 items	PROD-QOH=PROD-QOH-30

The table below shows the serial execution of the transactions under normal circumstances yielding the correct result.

TIME	TRANSACTION	STEP	STORED
1	T1	Read PROD-QOH	35
2	T1	PROD-QOH=35+100	
3	T1	Write PROD-QOH	135
4	T2	Read PROD-QOH	135
5	T2	PROD-QOH=135-30	
6	T2	Write PROD-QOH	105

But suppose that a transaction is able to read a PROD-QOH value from the table before a previous transaction has committed. The table below shows how the lost update problem can arise. Note that the first transaction T1 has not been committed when the second transaction T2 is executed

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	READ PROD-QOH	35
2	T2	READ PROD-QOH	35
3	T1	PROD-QOH=35+100	
4	T2	PROD-QOH=35-5	
5	T1	WRITE PROD-QOH	135
6	T2	WRITE PROD-QOH	5

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 has already accessed the uncommitted data. The table below shows the correct execution of transactions

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	READ PROD-QOH	35
2	T1	PROD-QOH=35+100	
3	T1	WRITE PRO-QOH	135
4	T1	ROLLED BACK	35
5	T2	READ PROD-QOH	35
6	T2	PROD-QOH=35-30	
7	T2	WRITE PROD-QOH	5

Inconsistent Retrievals

Inconsistent retrievals occur when a transaction accesses data before and after another transaction(s) 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 they are changed and other data after they are changed, thereby yielding inconsistent results.

The Scheduler

The scheduler is a special DBMS process that 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. 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. 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).

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.

Additionally, the scheduler facilitates data isolation to ensure that two transactions do not update the same data element at the same time

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 and 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).

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 Tl is being executed. This level of locking is good for batch processes, but it is unsuitable for multiuser DBMSs. You can imagine how slow the data access would be if thousands of transactions had to wait for the previous transaction to be completed before the next one could reserve the entire database.

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.

Page level

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 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 multiuser DBMS locking method.

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.

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.

Lock Types

Binary Locks

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. Therefore, every transaction requires a lock and unlock operation for each data item that is accessed. Such operations are automatically managed and scheduled by the DBMS therefore the user does not need to be concerned about locking or unlocking data items. Every DBMS has a default locking mechanism. If the end user wants to override the default, the LOCK TABLE and other SQL commands are available for that purpose.

However, binary locks are now considered too restrictive to yield optimal concurrency conditions. For example, the DBMS will not allow two transactions to read the same database object even though neither transaction updates the database, and therefore, no concurrency problems can occur.

Shared/Exclusive Locks

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:

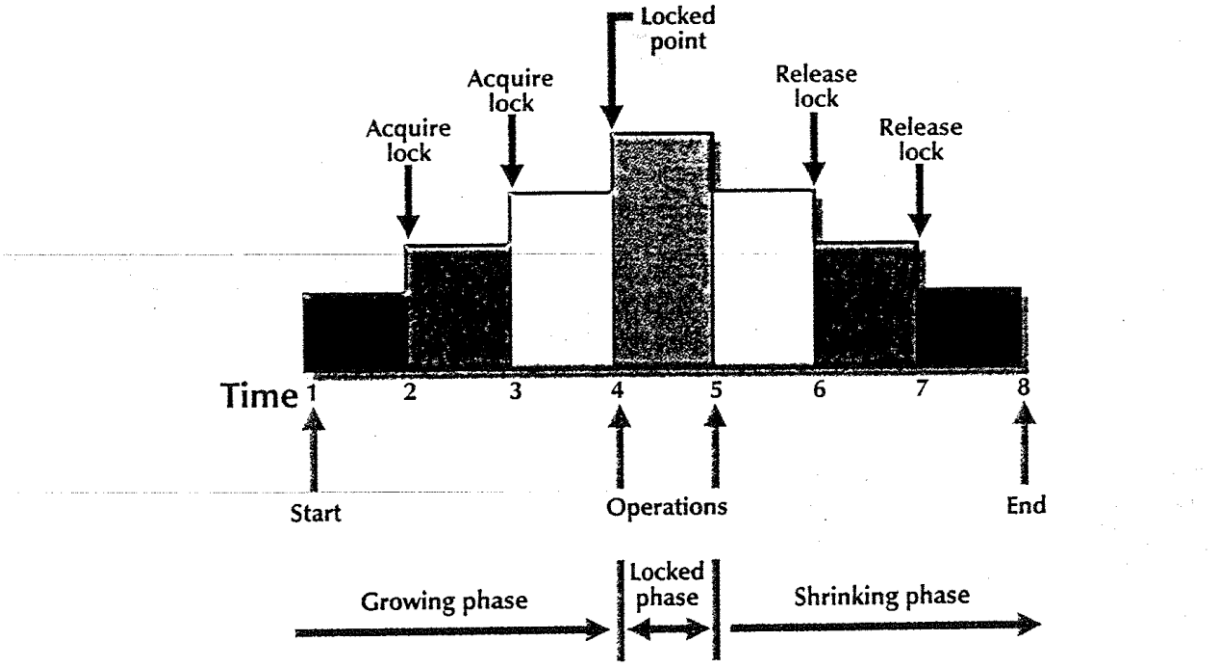
- unlocked,
- shared (read),
- exclusive (write).

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.
The exclusive lock is granted if and only if no other locks are held on the data item. 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. This condition is known as the **mutual exclusive rule**: only one transaction at a time can own an exclusive lock on the same object.
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 equivalent 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 that is serializability is guaranteed through a locking protocol known as two-phase locking, and deadlocks can be managed by using deadlock detection and prevention techniques.

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.
Two-phase locking increases the transaction processing cost and might cause additional undesirable effects. One undesirable effect is the possibility of creating deadlocks.



Deadlocks

A deadlock occurs when two or more 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.

The three basic techniques to control deadlocks are:

- Deadlock prevention. A transaction requesting a new lock is aborted when there is the possibility that adeadlock 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.

The choice of the best deadlock control method to use depends on the database environment. For example, if the probability of deadlocks is low, deadlock detection is recommended. However, if the probability of deadlocks is high, deadlock prevention is recommended. If response time is not high on the system's priority list, deadlock avoidance might be employed. All current DBMSs support deadlock detention in transactional databases, while

some DBMSs use a blend of prevention and avoidance techniques for other types of data, such as data warehouses or XML data.