

Transaction & Concurrency control

Transaction is defined as the sequence of server operation that is guaranteed by the server to be atomic in the presence of multiple clients and servers. Generally, hosts provide resources, and a transaction manager is responsible for developing and handling the transaction. Like any other transaction, a distributed transaction should include all four ACID properties (atomicity, consistency, isolation, durability).

A nested transaction is a transaction that is created inside another transaction. A nested transaction's purpose is to support transactions in stored procedures that can be called from a process already in a transaction or from a process that has no active transaction.

ACID (atomicity, consistency, isolation, and durability)

Atomicity. In a transaction involving two or more discrete pieces of information, either all of the pieces are committed or none are. This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

Consistency. A transaction either creates a new and valid state of data, or, if any failure occurs, returns all data to its state before the transaction was started.

Isolation. A transaction in process and not yet committed must remain isolated from any other transaction.

Durability. Committed data is saved by the system such that, even in the event of a failure and system restart, the data is available in its correct state.

Concurrency control manages simultaneous access to a resource. It prevents two users from accessing, editing the same resources at the same time and also serializes transactions for backup and recovery. The general area of concurrency control provides rules, methods, design methodologies, and theories to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system.

NEED

If transactions are executed serially, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable results may occur, such as:

- The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to

read the first value. The transactions that have read the wrong value end with incorrect results.

- The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
- The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Concurrency control protocols can be broadly divided into two categories –

- Lock based protocols
- Time stamp based protocols

Lock-based Protocols

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds –

Binary Locks – A lock on a data item can be in two states; it is either locked or unlocked.

Shared/exclusive – This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

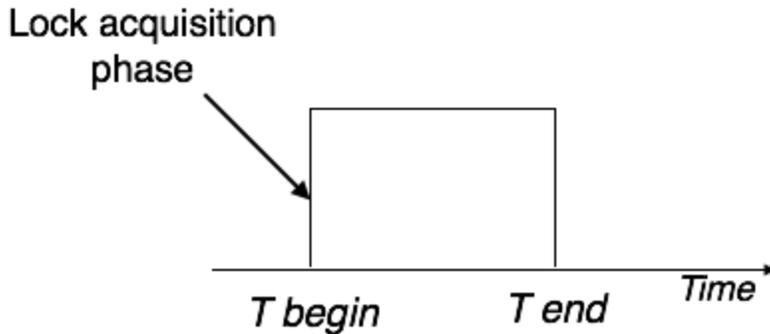
There are four types of lock protocols available –

- **Simplistic Lock Protocol**

Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the 'write' operation.

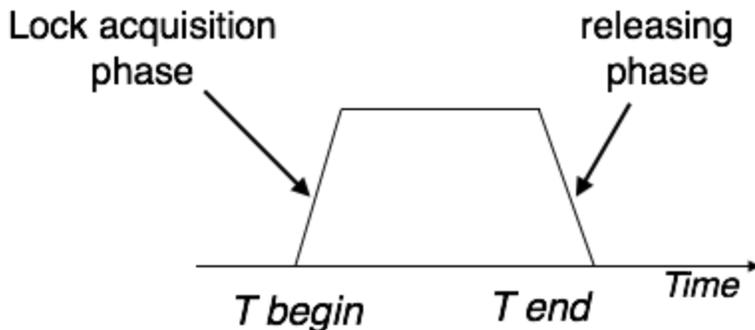
- **Pre-claiming Lock Protocol**

Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.



- **Two-Phase Locking 2PL**

This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.

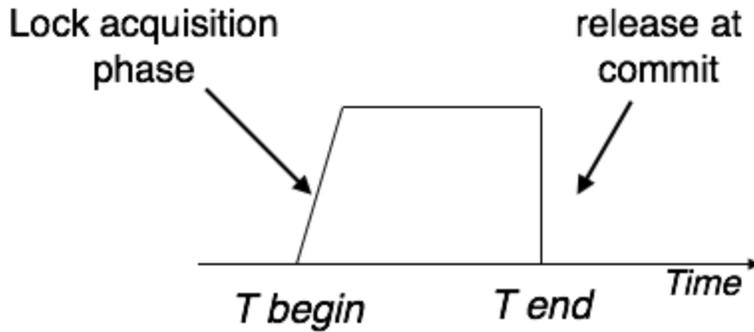


Two-phase locking has two phases, one is growing, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.

To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

- **Strict Two-Phase Locking**

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



Strict-2PL does not have cascading abort as 2PL does.

Timestamp-based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last 'read and write' operation was performed on the data item.

Timestamp Ordering Protocol

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction T_i is denoted as $TS(T_i)$.
- Read time-stamp of data-item X is denoted by $R\text{-timestamp}(X)$.
- Write time-stamp of data-item X is denoted by $W\text{-timestamp}(X)$.

Timestamp ordering protocol works as follows –

If a transaction T_i issues a read(X) operation –

1. If $TS(T_i) < W\text{-timestamp}(X)$
Operation rejected.
2. If $TS(T_i) \geq W\text{-timestamp}(X)$
Operation executed.
3. All data-item timestamps updated.

If a transaction T_i issues a write(X) operation –

1. If $TS(T_i) < R\text{-timestamp}(X)$
Operation rejected.
2. If $TS(T_i) < W\text{-timestamp}(X)$
Operation rejected and T_i rolled back.
3. Otherwise, operation executed.

Thomas' Write Rule

This rule states if $TS(T_i) < W\text{-timestamp}(X)$, then the operation is rejected and T_i is rolled back. Time-stamp ordering rules can be modified to make the schedule view serializable. Instead of making T_i rolled back, the 'write' operation itself is ignored.

[source : <http://people.cs.aau.dk/~bnielsen/ITEV-DistrSys-06/material/itev-transactions.pdf>]