CHAPTER - 10

Transaction Processing and Concurrency Control

10.1 Introduction to Transactions

Transaction is a sequence of read and write operations on the database with embedded database access queries that maintains the consistent and reliable computation. Thus, transaction management deals with the problem of always keeping the database in a consistent state even when concurrent accesses and failure occurs. In other word, the term *transaction* refers to a collection of operations that form a single logical unit of work. For *example*, transfer of money from one account to another is a transaction consisting of two updates, one to each account.

Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language (for example, C++, or Java), with embedded database accesses in *Java Database Connectivity (JDBC)* for Java application database access or *Open Database Connectivity* (ODBC) for Microsoft application database access . The transaction consists of all operations executed between the **begin transaction** and **end transaction**. This collection of steps must appear to the user as a single, indivisible unit.

It is important that either all actions of a transaction be executed completely, or, in case of some failure, partial effects of each incomplete transaction be undone. This property is called *atomicity* or *all-or-nothing property*. Further, once a transaction is successfully executed, its effects must persist in the database—a system failure should not result in the database forgetting about a transaction that successfully completed. This property is called *durability*.

In a database system where multiple transactions are executing concurrently, if updates to shared data are not controlled there is potential for transactions to see inconsistent intermediate states created by updates of other transactions. Such a situation can result in erroneous updates to data stored in the database. Thus, database systems must provide mechanisms to isolate transactions from the effects of other concurrently executing transactions. This property is called *isolation*.

Transaction State

- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction – only if no internal logical error
  - kill the transaction
- **Committed,** after *successful completion*.

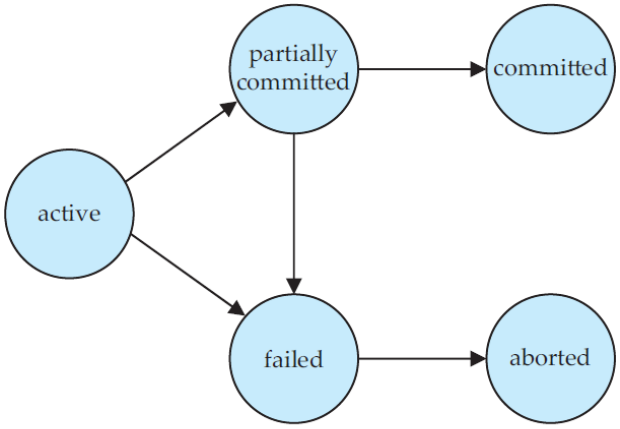


Figure State diagram of a transaction.

A transaction consists of sequence of query or updateable statements. SQL standard specifies, the transaction begins implicitly when SQL statement is executed and one of the following SQL statement must end with transaction commands.

- **COMMIT:** It commits (save) the current transaction changes on database / table/s by update statements. After the transaction is committed, a new transaction is automatically started.

- **ROLLBACK:** It rollback (undo) the current transaction. That is, it undo all the update performed by SQL statements. Thus database state is restored to what it was before the first statements of the transaction were executed.

If program terminates without executing either of the commands commit or rollback. The updates or changes to database are either committed or rollback. This depends upon SQL implementation. In many SQL implementations, if transactions are continued and at the same moment if the system is restarted or fails then transaction is rollback.

10.2 ACID Properties of Transactions

The consistency and reliability aspects of transactions are due to four properties: atomicity, consistency, isolation, and durability. Together, these are commonly referred to as the ACID properties of transactions. They are not entirely independent of each other; usually there are dependencies among them.

A. Atomicity: Atomicity refers to the fact that a transaction is treated as a unit of operation. Therefore, either all the transaction's actions are completed, or none of them are. This is also known as the "all-or-nothing property." Atomicity requires that if the execution of a transaction is interrupted by any sort of failure, the DBMS will be responsible for determining what to do with the transaction upon recovery from the failure. There are two types of failures.

a. Transaction recovery: Transaction recovery is maintaining transaction atomicity in the presence of failure on transaction operation due to to input data errors, deadlocks, or other factors. In these cases either the transaction aborts itself, or the DBMS may abort it while handling deadlocks.

b. Crash recovery: Crash recovery is ensuring transaction atomicity in the presence of system crashes such as media failures, processor failures, communication link breakages, power outages, and so on.

For example: State before the execution of transaction T_i

- The value of A = 50,000
- The value of B = 100

Then,

The failure occur (ex. Hardware failure)

- Failure happen after the **WRITE(A)** operation
- (at this moment A = 50000 – 5000 = 45000)
- And the value of B = 100 **(inconsistency state)**
- **In consistency state A = 45000 and B = (5100)**

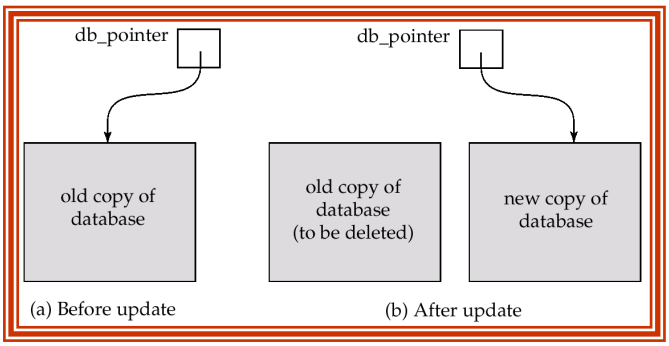
Idea behind ensuring atomicity is following:

- The database system keeps track of the old values of any data on which a transaction performs a write
- If the transaction does not complete, the DBMS restores the old values to make it appear as though the transaction have never execute.

B. Durability: Durability refers to that property of transactions which ensures that once a transaction commits, its results are permanent and cannot be erased from the database. Therefore, the DBMS ensures that the results of a transaction will survive subsequent system failures. The durability property brings forth the issue of database recovery, that is, how to recover the database to a consistent state where all the committed actions are reflected.

Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.
- The *shadow-database* scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.



C. Consistency: The consistency of a transaction is simply its correctness. In other words, a transaction is a correct program that maps one consistent database state to another. Verifying that transactions are consistent is the concern of integrity enforcement. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. There are two advantages of concurrency that are:

- **Increased processor and disk utilization**, leading to better transaction *throughput* (one transaction can be using the CPU while another is reading from or writing to the disk) and
- **Reduced average response time** for transactions (short transactions need not wait behind long ones).

D. Isolation: Isolation is the property of transactions that requires each transaction to see a consistent database at all times. In other words, an executing transaction cannot reveal its results to other concurrent transactions before its commitment. There are a number of reasons for insisting on isolation. One has to do with maintaining the inter-consistency of transactions. If two concurrent transactions access a data item that is being updated by one of them, it is not possible to guarantee that the second will read the correct value.

Example: Consider the following two concurrent transactions (T1 and T2), both of which access data item x. Assume that the value of x before they start executing is 50.

T_1	T_2
Read(x)	Read(x)
$x \leftarrow x+1$	$x \leftarrow x+1$
Write(x)	Write(x)
Commit	Commit

The following is one possible sequence of execution of the actions of these transactions:

Case-I:

T_1: Read(x)
T_1: $x \leftarrow x+1$
T_1: Write(x)
T_1: Commit
T_2: Read(x)
T_2: $x \leftarrow x+1$
T_2: Write(x)
T_2: Commit

Case-II:

T_1: Read(x)
T_1: $x \leftarrow x+1$
T_2: Read(x)
T_1: Write(x)
T_2: $x \leftarrow x+1$
T_2: Write(x)
T_1: Commit
T_2: Commit

- In case-I, there are no problems; transactions T1 and T2 are executed one after the other and transaction T2 reads 51 as the value of x. Note that if, instead, T2 executes before T1, T2 reads 51 as the value of x. So, if T1 and T2 are executed one after the other (regardless of the order), the second transaction will read 51 as the value of x and x will have 52 as its value at the end of execution of these two transactions.

- In case-II, transaction T2 reads 50 as the value of x. This is incorrect since T2 reads x while its value is being changed from 50 to 51. Furthermore, the value of x is 51 at the end of execution of T1 and T2 since T2's Write will overwrite T1's Write.

10.3 Schedules and Serializability

Schedules (or *history*) is a sequences that indicate the chronological order in which instructions of concurrent transactions are executed. Often it is a *list* of operations (actions) ordered by time, performed by a set of transactions that are executed together in the system. The Scheduler is the special DBMS Program to establish the order of operations in which concurrent transactions are executes.

Types of schedule

A. Serial

The transactions are executed non-interleaved i.e. a serial schedule is one in which no transaction starts until a running transaction has ended. In this **example**, the horizontal axis represents the different transactions in the schedule D. The vertical axis represents time order of operations. Schedule D consists of three transactions T1, T2, T3. The schedule describes the actions of the transactions as seen by the DBMS. First T1 Reads and Writes to object X, and then Commits. Then T2 Reads and Writes to object Y and Commits, and finally T3 Reads and Writes to object Z and Commits. This is an example of a *serial* schedule, i.e., sequential with no overlap in time, because the actions of in all three transactions are sequential, and the transactions are not interleaved in time.

$$D = \begin{bmatrix} T1 & T2 & T3 \\ R(X) & & \\ W(X) & & \\ Com. & & \\ & R(Y) & \\ & W(Y) & \\ & Com. & \\ & & R(Z) \\ & & W(Z) \\ & & Com. \end{bmatrix}$$

Representing the schedule D above by a table (rather than a list) is just for the convenience of identifying each transaction's operations in a glance. This notation is used throughout the article below. A more common way in the technical literature for representing such schedule is by a list: `D = R1(X) W1(X) Com1 R2(Y) W2(Y) Com2 R3(Z) W3(Z) Com3`

B. Serializable

A schedule is equivalent to serializable is it is equivalent to a *serial schedule*. It relates to the *isolation* property of a database transaction. The basic assumption of serializability is preserves database consistency by each transaction.
In schedule E, the order in which the actions of the transactions are executed is not the same as in D, but in the end, E gives the same result as D (in above example).

$$E = \begin{bmatrix} T1 & T2 & T3 \\ R(X) & & \\ & R(Y) & \\ & & R(Z) \\ W(X) & & \\ & W(Y) & \\ & & W(Z) \\ Com. & Com. & Com. \end{bmatrix}$$

Conflict Serializability

Instructions *li* and *lj* of transactions *Ti* and *Tj* respectively, **conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q*.

- *li* = **read**(*Q*), *lj* = **read**(*Q*). *li* and *lj* don't conflict.
- *li* = **read**(*Q*), *lj* = **write**(*Q*). They conflict.
- *li* = **write**(*Q*), *lj* = **read**(*Q*). They conflict
- *li* = **write**(*Q*), *lj* = **write**(*Q*). They conflict

A schedule is said to be conflict-serializable when the schedule is conflict-equivalent to one or more serial schedules. In example, G is conflict-equivalent to the serial schedule <T1,T2>, but not <T2,T1>.

$$G = \begin{bmatrix} T1 & T2 \\ R(A) & \\ & R(A) \\ W(B) & \\ Com. & \\ & W(A) \\ & Com. \end{bmatrix}$$

C. Recoverable

Transactions commit only after all transactions whose changes they read, commit. In figure, F is recoverable because T1 commits before T2, that makes the value read by T2 correct. Then T2 can commit itself. In F2, if T1 aborted, T2 has to abort because the value of A it read is incorrect. In both cases, the database is left in a consistent state.

$$F = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Com. & \\ & Com. \end{bmatrix} \quad F2 = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Abort & \\ & Abort \end{bmatrix}$$

Unrecoverable

If a transaction T1 aborts, and a transaction T2 commits, but T2 relied on T1, we have an unrecoverable schedule. In this example, G is unrecoverable, because T2 read the value of A written by T1, and committed. T1 later aborted, therefore the value read by T2 is wrong, but since T2 committed, this schedule is unrecoverable.

$$G = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ & Com. \\ Abort & \end{bmatrix}$$

10.4 Concurrency Control

When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.

To ensure serializability, we can use various concurrency-control schemes. All these schemes either delay an operation or abort the transaction that issued the operation.

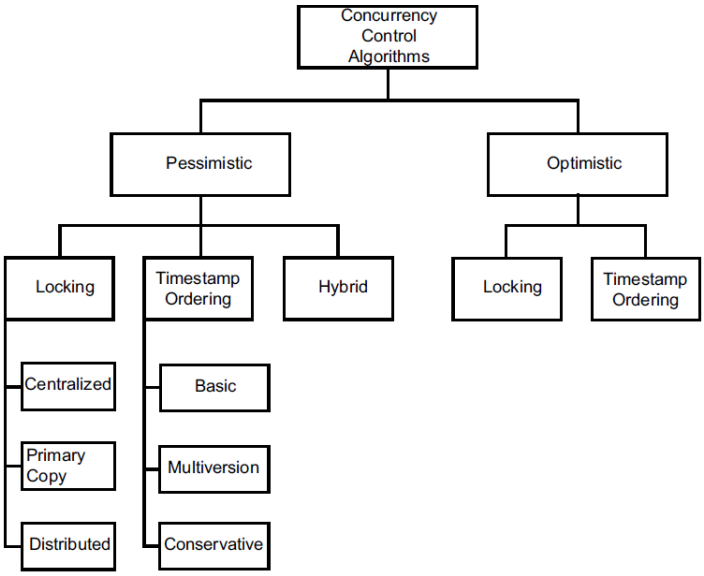


Fig. Classification of Concurrency Control Algorithms

A. Locking-Based Concurrency Control Algorithms

The main idea of locking-based concurrency control is to ensure that a data item that is shared by conflicting operations is accessed by one operation at a time. This is accomplished by associating a "lock" with each lock unit. This lock is set by a transaction before it is accessed and is reset at the end of its use. Obviously a lock unit cannot be accessed by an operation if it is already locked by another. Thus a lock request by a transaction is granted only if the associated lock is not being held by any other transaction.

In locking-based systems, the scheduler is a lock manager (LM). The transaction manager passes to the lock manager the database operation (read or write) and associated information (such as the item that is accessed and the identifier of the transaction that issues the database operation). The lock manager then checks if the lock unit that contains the data item is already locked. If so, and if the existing lock mode is incompatible with that of the current transaction, the current operation is delayed. Otherwise, the lock is set in the desired mode and the database operation is passed on to the data processor for actual database access. The transaction manager is then informed of the results of the operation. The termination of a transaction results in the release of its locks and the initiation of another transaction that might be waiting for access to the same data item.

There are two type of locks (or lock modes) which are:

- **Read lock:** Any transaction (T) read a data item if it has read lock.
- **Write lock:** Any transaction (T) write a data item if it has write lock.

The DBMS not only manages locks but also handles the lock management responsibilities on behalf of the transactions. In other word, the user do not need to specify when data need to be locked.

Two-phase locking (2PL)

The two-phase locking rule simply states that no transaction should request a lock after it releases one of its locks. Alternatively, a transaction should not release a lock until it is certain that it will not request another lock. 2PL algorithms execute transactions in two phases. Each transaction has a growing phase, where it obtains locks and accesses data items, and a shrinking phase, during which it releases locks. The lock point is the moment when the transaction has achieved all its locks but has not yet started to release any of them. Thus the lock point determines the end of the growing phase and the beginning of the shrinking phase of a transaction.

The two-phase locking protocol allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item. The protocol ensures serializability, but not deadlock freedom. In the absence of information concerning the manner in which data items are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability.

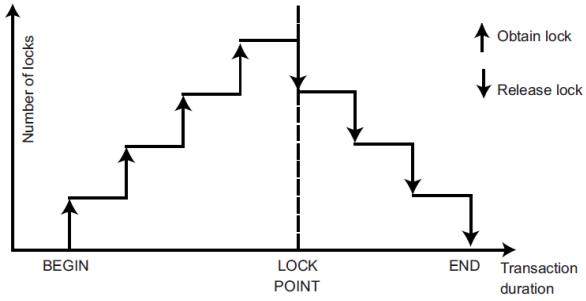


Fig. 2PL Lock Graph

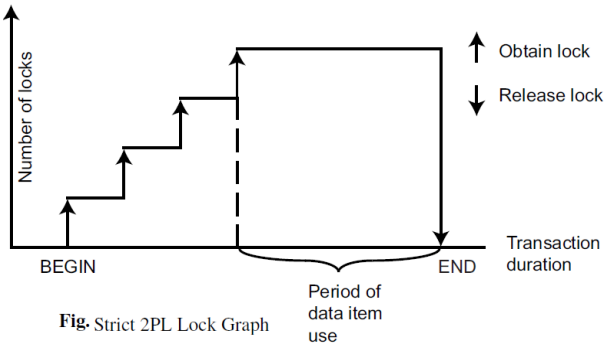


Fig. Strict 2PL Lock Graph

The first figure indicates that the lock manager releases locks as soon as access to that data item has been completed. This permits other transactions awaiting access to go ahead and lock it, thereby increasing the degree of concurrency. However, this is difficult to implement since the lock manager has to know that the transaction has obtained all its locks and will not need to lock another data item. The lock manager also needs to know that the transaction no longer needs to access the data item in question, so that the lock can be released. Finally, if the transaction aborts after it releases a lock, it may cause other transactions that may have accessed the unlocked data item to abort as well. This is known as cascading aborts. These problems may be overcome by strict two-phase locking, which releases all the locks together when the transaction terminates (commits or aborts).