4. Consistency and Transactions

DB operations change the internal state of the database, thus describing a situation in the real world that differs from the previous one. In fact, not all states of a database might reflect a real "consistent" state of the real world – some might be inconsistent.

Example: A banking application needs to transfer funds from one account to another. This would mean that the balance of one account must be decremented while the balance of the other account has to be incremented by the same amount. Executing only one of these operations and not both would leave the database in an inconsistent state.

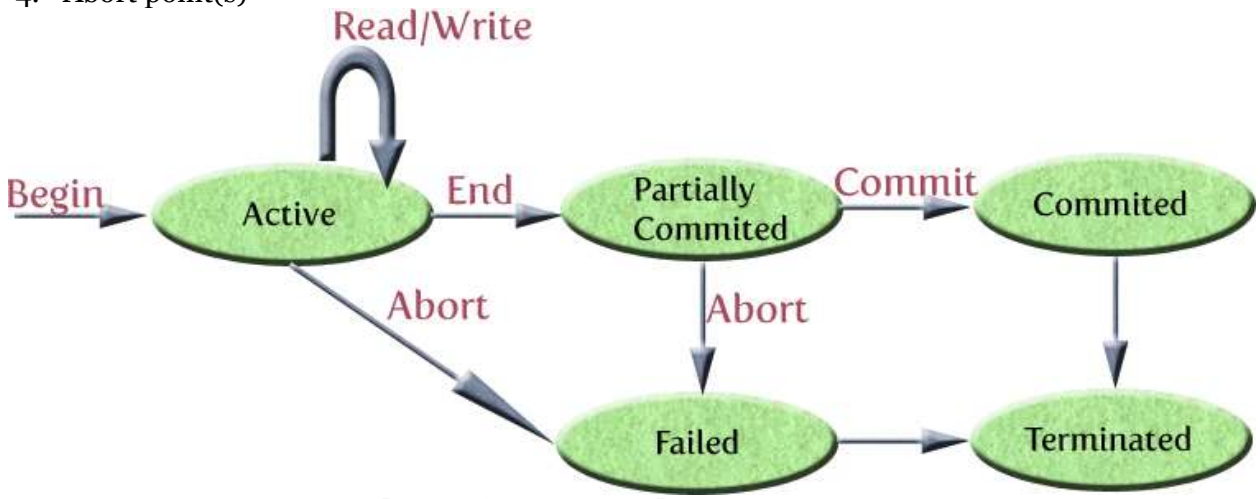
4.1 The Transaction Concept

In database systems and - as we will see later - in general distributed systems, operations will be "bundled" into sequences that guarantee that consistent database states are transformed into consistent database states. Such a bundle of operations is referred to as a **transaction**. Transactions are characterized by the so-called ACID principle [http://en.wikipedia.org/wiki/ACID].

Transactions protect the database from irregularities caused by failure of operations and the need for concurrent access. For the sake of simplicity transaction theory only considers read and write operations, other – like calculations – are not regarded as relevant to the consistency of the database. The following diagram describes the main mechanism behind a transaction:

A transaction is characterized by several points in time:

1. Begin of transaction
2. End of transaction
3. Commit point
4. Abort point(s)



Transaction State Diagram [Elmasri, Navathe 03]

A transaction is dynamically enclosed into a pair of special statements, “Begin of transaction” tells the DBMS that a transaction is about to start, “Commit” declares the transaction to be correctly performed, while “Abort” describes a condition under which the transaction cannot be succesfully continued.

When a transaction is commited, the modifications are made permanent in the database, when it's aborted, measures might be taken to undo operations that have already been made – the transaction is “rolled back”.[1]

Reasons for an abort are quite a few:

- Hardware errors: including power fails, disk errors, network errors.
- Software errors: like errors/exceptions inside the transaction code, programming errors.
- Logical problems detected during the transaction.
- Aborts enforced by concurrency control: especially to remove deadlocks.

4.2 The need for Concurrency

From the viewpoint of transactions a DBMS is nothing more that a Transaction Processing System (TPS), i.e. an algorithm that keeps track of transactions that have to executed and makes sure that they're executed correctly.

Technically, if all the transactions currently ready for execution were independent – and we can safely assume they are – executing one transaction after the other would be a safe way to process them. Unfortunately, since some of the operations might have to wait until an I/O operation is completed or be suspended for any other reason. Another problem occurs with “long transactions”, i.e. transactions that might take hours, days, weeks or longer to finish. An example of such a long transaction is found in CAD (Computer Aided Design) systems where the representation of the design object is stored in a database and protected against concurrent modifications until it is committed by the main designer.

4.2.1 Schedules and Serializability

Since transactions consist of a sequence of database operations, concurrent execution of two or more transactions can have different outcomes. To illustrate, let's assume a set of transactions numbered from 1 to n, each executing a sequence of the following operations:

- $r_n(x)$: transaction n reads the value of x
- $w_n(x)$: transaction n writes a new value of x
- $abort_n$: transaction n is aborted
- $commit_n$: transaction n is commited

1 Technically a transaction could write all updates into a special area, called “shadow” and after committing copy all over into the main “live” database system. Yet, we assume here that the DBMS actually writes all updates into the live database and in case of an abort rolls the database back into the previous state from a log file [Elmasri, Navathe 03].

A TPS would now have to schedule these transactions to execute these operations interleaved, so that

1. An operation within a given transaction is never scheduled before all the preceding operations.
2. Otherwise, the sequence of operations is undetermined.

Any sequence of operations from a set of transactions is called a **schedule**. Constructing a "good" schedule is one of the most important tasks for a TPS. Without interleaving a schedule would execute the given transactions one after the other, a **serial** schedule. If an interleaved schedule could be found that's equivalent to a serial schedule this would be a **serializable** schedule.

If more than one transaction would be executed interleaved, some unwanted effects could happen:

- **Lost Update**: $r_1(x) \rightarrow r_2(x) \rightarrow w_1(x) \rightarrow w_2(x)$
  T_1 and T_2 read the same value, T_1 writes the modified value back first, then T_2 – the update made by T_1 is lost.
- **Unrepeatable Read**: $r_1(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_1(x)$
  T_1 and T_2 read the same value, T_2 writes the modified value back and T_1 – assuming it could just read the old value back - reads the new value instead.
- **Dirty Read**: $r_1(x) \rightarrow w_1(x) \rightarrow r_2(x) \rightarrow abort_1$
  T_1 stores a value in the database and later aborts (and rolls back); in the meantime T_2 reads this invalid value.

Phantom Reads don't really fit this scheme; they occur if T_1 has executed a SELECT over a table, T_2 then inserts another row into this table and when T_1 re-executes the query a new row will show up in the result – the "phantom".

4.2.2 Locking

In order to combat these phenomena the main idea is to **lock** certain parts of the database – thereby granting access to these data items only to transactions that are allowed to do so.

Locks are set by the DBMS lock manager on the request of a transaction. In the simplest case, a **binary lock** will only let at most one transaction – the one that has set (or "holds") the lock - have access to the item, until the lock is released. Since this scheme is very restrictive, the idea of distinguishing between **read** and **write locks** (or "shared" and "exclusive" locks) allows for a better utilization of the DB item, by allowing all transactions to read the item, unless a transaction writes it using a write lock.

Acquiring these locks must follow an underlying policy from the transactions involved:

1. A transaction must set a read (write) lock on x before executing a read (write) operation on x.
2. A transaction must not lock an item twice.
3. A transaction must unlock x after all read/write operations on x are performed.
4. A transaction will only unlock items it has locked before.

In fact, rule 2 could be relaxed in a way that a read lock might be converted to a write lock (upgrading) or vice versa (downgrading).

The rules for locks are pretty simple:

- If a transaction holds a read lock on an item, other transactions can acquire a read lock for this item, but no write lock – hence the term "shared".
- If a transaction holds a write lock on an item no other transaction can acquire a lock on it, neither read or write – hence the term "exclusive".

4.2.3 Two-Phase Locking

Interestingly enough, this mechanism is *not* sufficient to secure serializability [Elmasri, Navathe 03]:

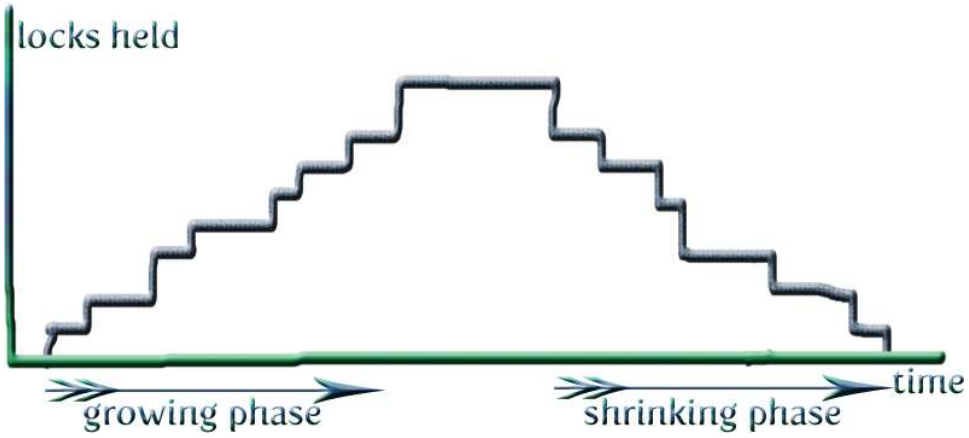
T_1	T_2
read-lock$_1$(y)	read-lock$_2$(x)
read$_1$(y)	read$_2$(x)
unlock$_1$(y)	unlock$_2$(x)
write-lock$_1$(x)	write-lock$_2$(y)
read$_1$(x)	read$_2$(y)
x = x + y	y = x+y
write$_1$(x)	write$_2$(y)
unlock$_1$(x)	unlock$_2$(y)

With initial values x=20 and y=30, the results for the two serial schedules would be: T_1T_2 : x=50, y=80 and T_2T_1 : x=70, y=50. The following schedule, though, results in x=50 and y=50:

T_1	T_2
read-lock$_1$(y)	
read$_1$(y)	
unlock$_1$(y)	
	read-lock$_2$(x)
	read$_2$(x)
	unlock$_2$(x)
	write-lock$_2$(y)
	read$_2$(y)
	y = x+y
	write$_2$(y)
	unlock$_2$(y)
write-lock$_1$(x)	
read$_1$(x)	
x = x + y	
write$_1$(x)	
unlock$_1$(x)	

After studying the effect for a moment, it becomes clear that T_1 should not have released the lock on y so early and neither should T_2 release its lock on x so early. A more restrictive policy to prevent this would have to follow the rule:

All locking operations (either read or write) must precede the first unlock operation.



Two-Phase Lock Protocol

So, in the first phase the number of locks held by the transaction will grow, then DB operations are performed and then – in the second phase – the locks will be released, making their number shrink.

The **Two-Phase Lock Protocol** (2PL) imposes this constraint on how locks are acquired and released. The 2PL though does not prevent **deadlocks** or **starvation.**

A deadlock occurs when two transactions are waiting for each other because one transaction has to release a lock the other one wants to acquire and vice versa. Deadlocks can be avoided, but a high cost, so deadlock detection and removal are the weapons of a DBMS to combat them.

Starvation can occur as a consequence of an unfair scheduling algorithm inside the DBMS. A transaction might have to wait for a lock for an indefinite amount of time, while other transactions are processed normally. Scheduling policies that avoid starvation are based on a First-In-First-Out basis or use priority-based scheduling with aging.

Multi-phase protocols like 2PL occur relatively frequently in the theory of databases and distributed computing. Later (in chapter 4) we will meet a protocol suitable for distributed databases, the Two-Phase Commit Protocol.