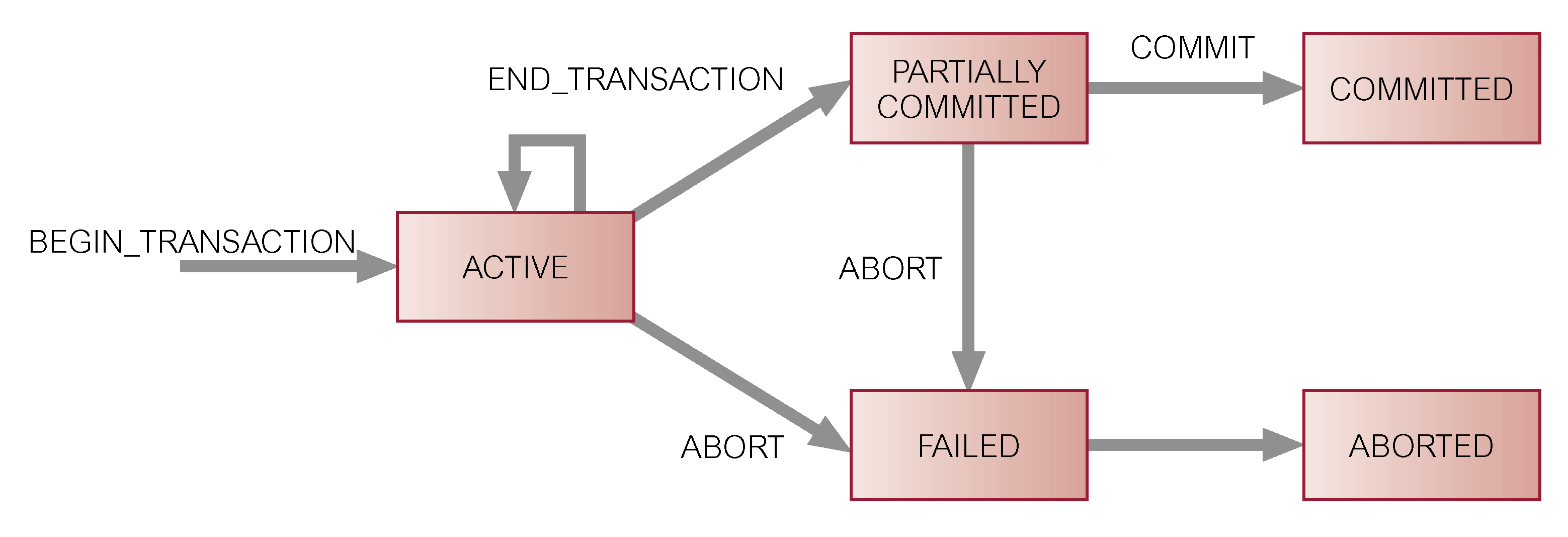
**Transaction Management**

**Transaction**

Action, or series of actions, carried out by user or application, which reads or updates contents of database.

* Logical unit of work on the database.
* Application program is series of transactions with non-database processing in between.
* Transforms database from one consistent state to another, although consistency may be violated during transaction.
* Can have one of two outcomes:
  + - Success - transaction *commits* and database reaches a new consistent state.
    - Failure - transaction *aborts*, and database must be restored to consistent state before it started.
    - Such a transaction is *rolled back* or *undone*.
* Committed transaction cannot be aborted.
* Aborted transaction that is rolled back can be restarted later.

**State Transition Diagram for Transaction**



**Properties of Transactions**

Four basic *(ACID)* properties of a transaction are:

* **Atomicity:**  ‘All or nothing’ property.
* **Consistency:** Must transform database from one consistent state to another.
* **Isolation**: Partial effects of incomplete transactions should not be visible to other transactions.
* **Durability:** Effects of a committed transaction are permanent and must not be lost because of later failure.

**Concurrency Control**

Process of managing simultaneous operations on the database without having them interfere with one another.

* Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
* Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

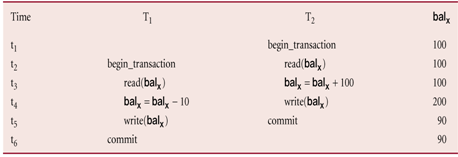
**Need for Concurrency Control**

Three examples of potential problems caused by concurrency:

* + Lost update problem.
  + Uncommitted dependency problem.
  + Inconsistent analysis problem.

**Lost Update Problem**

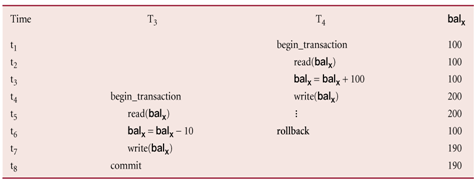
* Successfully completed update is overridden by another user.
* T1 withdrawing £10 from an account with balx, initially £100.
* T2 depositing £100 into same account.
* Serially, final balance would be £190.



* Loss of T2’s update avoided by preventing T1 from reading balx until after update.

**Uncommitted Dependency Problem**

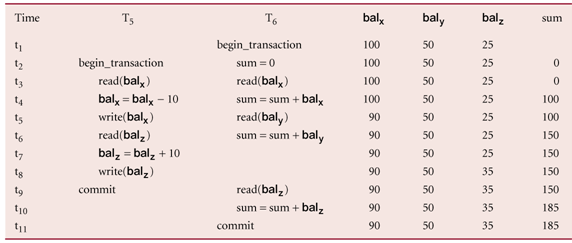
* Occurs when one transaction can see intermediate results of another transaction before it has committed.
* T4 updates balx to £200 but it aborts, so balx should be back at original value of £100.
* T3 has read new value of balx (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.



* Problem avoided by preventing T3 from reading balx until after T4 commits or aborts.

**Inconsistent Analysis Problem**

* Occurs when transaction reads several values but second transaction updates some of them during execution of first.
* Sometimes referred to as *dirty read* or *unrepeatable read*.
* T6 is totaling balances of account x (£100), account y (£50), and account z (£25).
* Meantime, T5 has transferred £10 from balx to balz, so T6 now has wrong result (£10 too high).



* Problem avoided by preventing T6 from reading balx and balz until after T5 completed updates.

**Serializability**

* Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference.
* Could run transactions serially, but this limits degree of concurrency or parallelism in system.
* Serializability identifies those executions of transactions guaranteed to ensure consistency.

**Schedule**

Sequence of reads/writes by set of concurrent transactions.

**Serial Schedule**

Schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions.

**Nonserial Schedule**

* Schedule where operations from set of concurrent transactions are interleaved.
* Objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.
* In other words, want to find nonserial schedules that are equivalent to *some* serial schedule. Such a schedule is called *serializable*.

**In serializability, ordering of read/writes is important:**

(a) If two transactions only read a data item, they do not conflict and order is not important.

(b) If two transactions either read or write completely separate data items, they do not conflict and order is not important.

(c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.

**Concurrency Control Techniques**

* Two basic concurrency control techniques:
  + Locking,
  + Timestamping.
* Both are conservative approaches: delay transactions in case they conflict with other transactions.
* Optimistic methods assume conflict is rare and only check for conflicts at commit.

**Locking**

Transaction uses locks to deny access to other transactions and so prevent incorrect updates.

* Most widely used approach to ensure serializability.
* Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) lock on a data item before read or write.
* Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

**Locking - Basic Rules**

* If transaction has shared lock on item, can read but not update item.
* If transaction has exclusive lock on item, can both read and update item.
* Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
* Exclusive lock gives transaction exclusive access to that item.
* Some systems allow transaction to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.

**Two-Phase Locking (2PL)**

Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.

Two phases for transaction:

* + Growing phase - acquires all locks but cannot release any locks.
  + Shrinking phase - releases locks but cannot acquire any new locks.

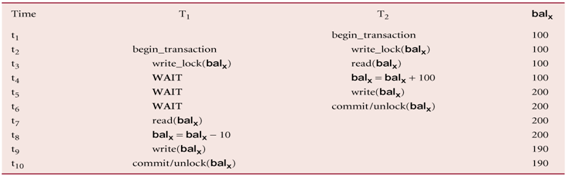
There is no requirement that all locks be obtained simultaneously. Normally, the transaction acquires some locks, does some processing, and goes on to acquire additional locks as needed. However, it never releases any lock until it has reached a stage where no new locks are needed. The rules are:

A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.

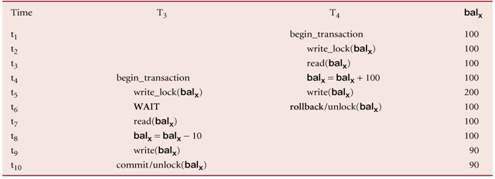
Once the transaction releases a lock, it can never acquire any new locks.

If upgrading of locks is allowed, upgrading can take place only during the growing phase and may require that the transaction wait until another transaction releases a shared lock on the item. Downgrading can take place only during the shrinking phase. We now look at how two-phase locking is used to resolve the three problems identified before:

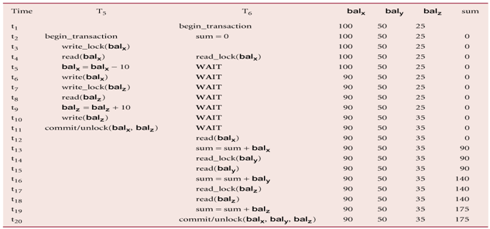
**Preventing Lost Update Problem using 2PL**



**Preventing Uncommitted Dependency Problem using 2PL**



**Preventing Inconsistent Analysis Problem using 2PL**



**Timestamping**

* Transactions ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.
* Conflict is resolved by rolling back and restarting transaction.
* No locks so no deadlock.

**Timestamp**

A unique identifier created by DBMS that indicates relative starting time of a transaction.

* Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.
* Read/write proceeds only if *last update on that data item* was carried out by an older transaction.
* Otherwise, transaction requesting read/write is restarted and given a new timestamp.
* Also timestamps for data items:
  + - read-timestamp - timestamp of last transaction to read item;
    - write-timestamp - timestamp of last transaction to write item.

With timestamping, if a transaction attempts to read or write a data item, then the read or write is only allowed to proceed if the last update on that data item was carried out by an older transaction. Otherwise, the transaction requesting the read/write is restarted and given a new timestamp. New timestamps must be assigned to restarted transactions to prevent their being continually aborted and restarted. Without new timestamps, a transaction with an old timestamp might not be able to commit owing to younger transactions having already committed.