Transaction,
Concurrency Control &
Recovery

Outline

Introduction to Transaction

A **transaction** is a *Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert/update/delete)*. E.g., transaction to transfer $50 from account A to account B:

1. **read_item**(*A*)
2. *A* := *A* – 50
3. **write_item**(*A*)
4. **read_item**(*B*)
5. *B* := *B* + 50
6. **write_item**(*B*)

read_item(X) : Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.

write_item(X) : Writes the value of program variable X into the database item named X.

- *A **transaction** (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.*

Introduction to Transaction

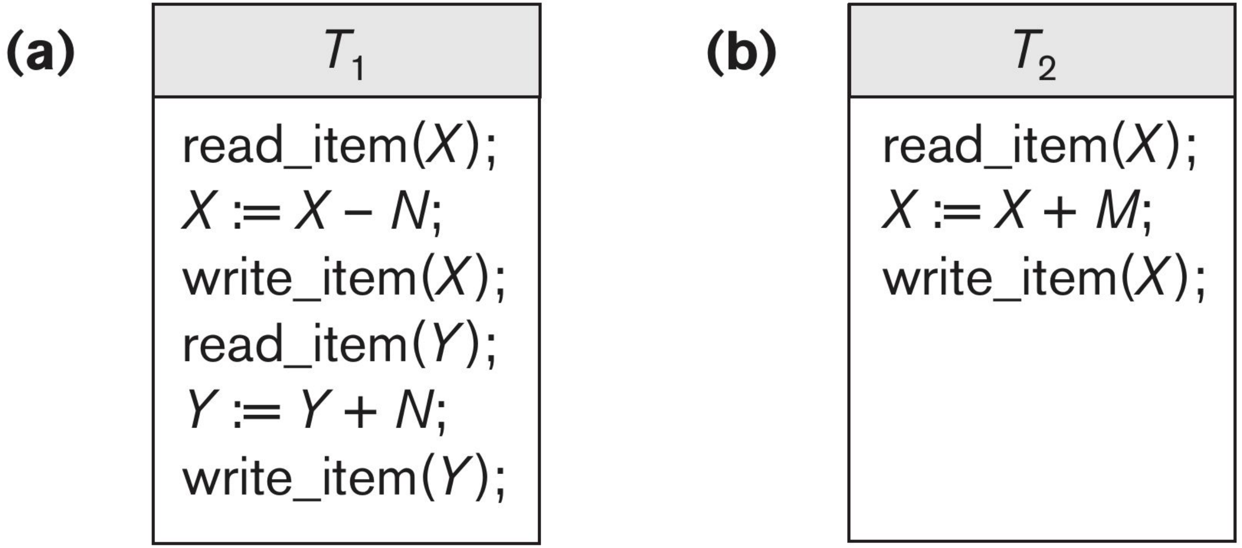
Executing a **read_item(X)** command includes the following steps:

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer). The size of the buffer is the same as the disk block size.
- Copy item X from the buffer to the program variable named X.

Executing a **write_item(X)** command includes the following steps:

- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Copy item X from the program variable named X into its correct location in the buffer.
- Store the updated disk block from the buffer back to disk (either immediately or at some later point in time).

Introduction to Transaction



(a)

T_1
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

(b)

T_2
read_item(X);
$X := X + M$;
write_item(X);

Figure : Two sample transactions. (a) Transaction T1. (b) Transaction T2.

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure the following properties often called the ACID properties; they should be enforced by the concurrency control and recovery methods of the DBMS.

- **Atomicity:** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

  Problem → End transaction

  - The atomicity property requires that we execute a transaction to completion. It is the responsibility of the **transaction recovery subsystem of a DBMS to ensure atomicity**. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

- **Consistency:** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

  No Problem → State 1 to State 2 Move on with the process

  - A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the complete execution of the transaction, assuming that no interference with other transactions occurs.

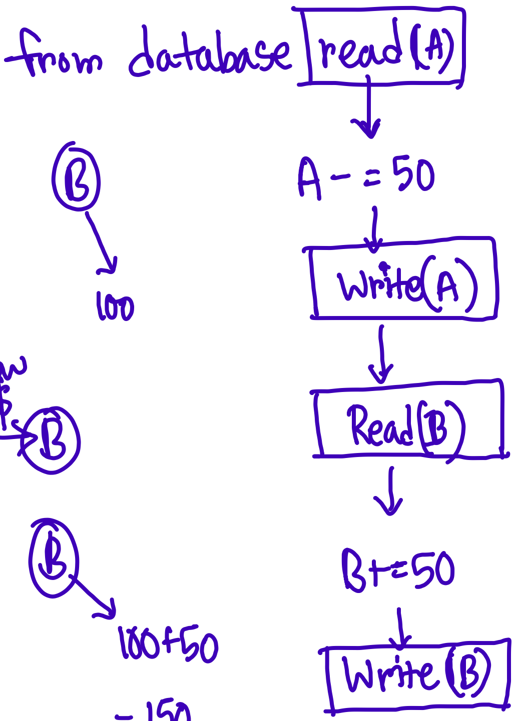
ACID Properties

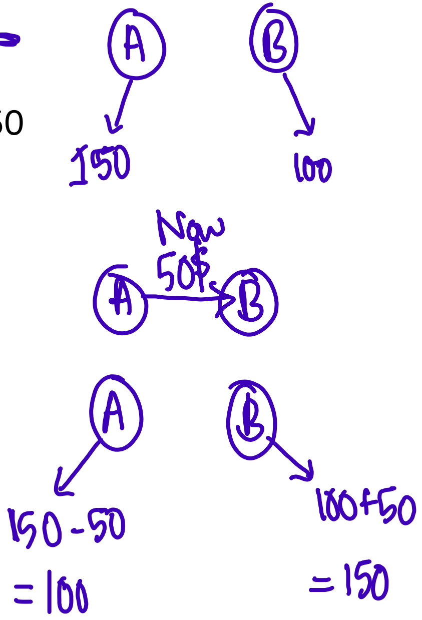
- **Isolation:** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
  - The isolation property is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems.
- **Durability or Permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.
  - The durability property is the responsibility of the recovery subsystem of the DBMS. Recovery protocols enforce durability and atomicity.

ACID Properties : Atomicity Requirement

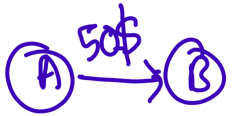
Transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)





- **Atomicity Requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database



ACID Properties : Durability Requirement

Transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B)*

- **Durability Requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

ACID Properties : Consistency Requirement

Transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Consistency Requirement in this example:
The sum of A and B is unchanged by the execution of the transaction.

In general, consistency requirements include,

- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints, e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent. Erroneous transaction logic can lead to inconsistency.

ACID Properties : Isolation Requirement

- **Isolation Requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

T1	T2
1. read(*A*) 2. *A* := *A* – 50 3. write(*A*)	
	read(A), read(B), print(A+B)
4. read(*B*) 5. *B* := *B* + 50 6. write(*B*)	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

Transaction Processing

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes - **Recovery**
- Concurrent execution of multiple transactions - **Concurrency Control**

Concurrency Control

Multiple transactions are allowed to run concurrently in the system.

Advantages are:

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

Concurrency control schemes – mechanisms to achieve isolation

- That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Why Concurrency Control Is Needed

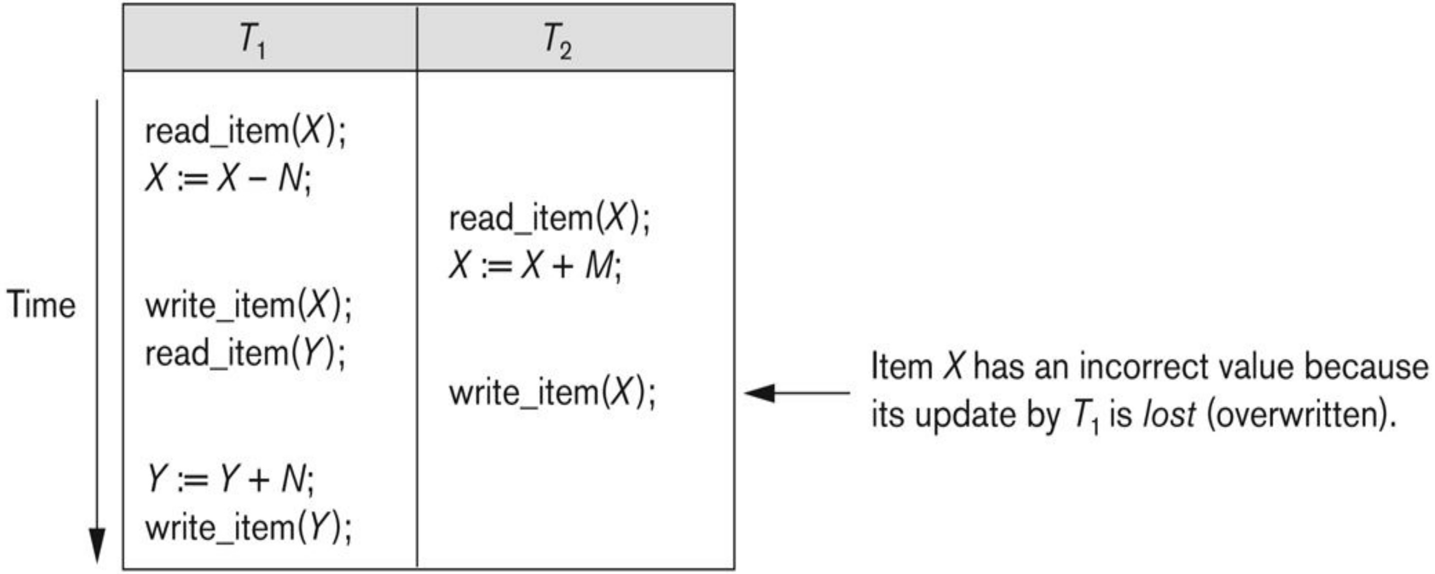
- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason
  - The updated item is accessed by another transaction before it is changed back to its original value.

- **The Incorrect Summary Problem**
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

Concurrency Control (Lost Update Problem)

Example of **Lost Update Problem** due to uncontrolled concurrent execution



T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
 read_item(Y);	

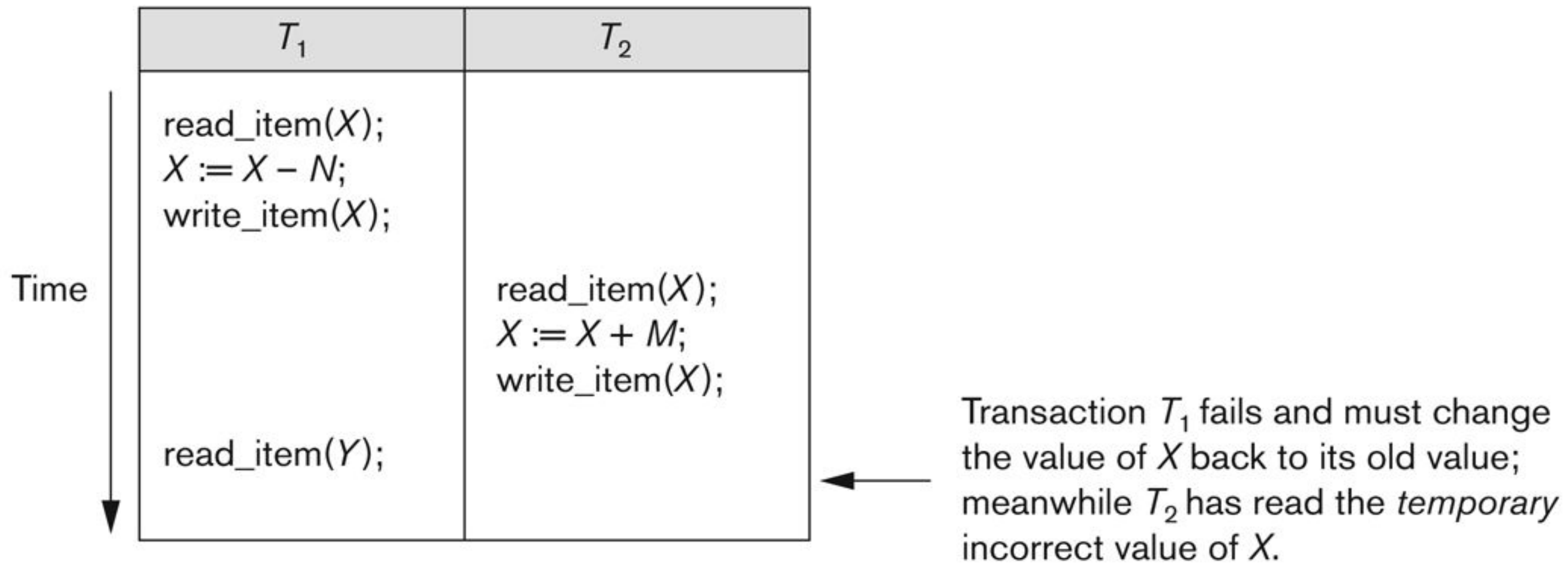
	write_item(X);
$Y := Y + N$; write_item(Y);	

Time

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Concurrency Control (Temporary Update Problem)

Example of **Temporary Update Problem** due to uncontrolled concurrent execution



T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

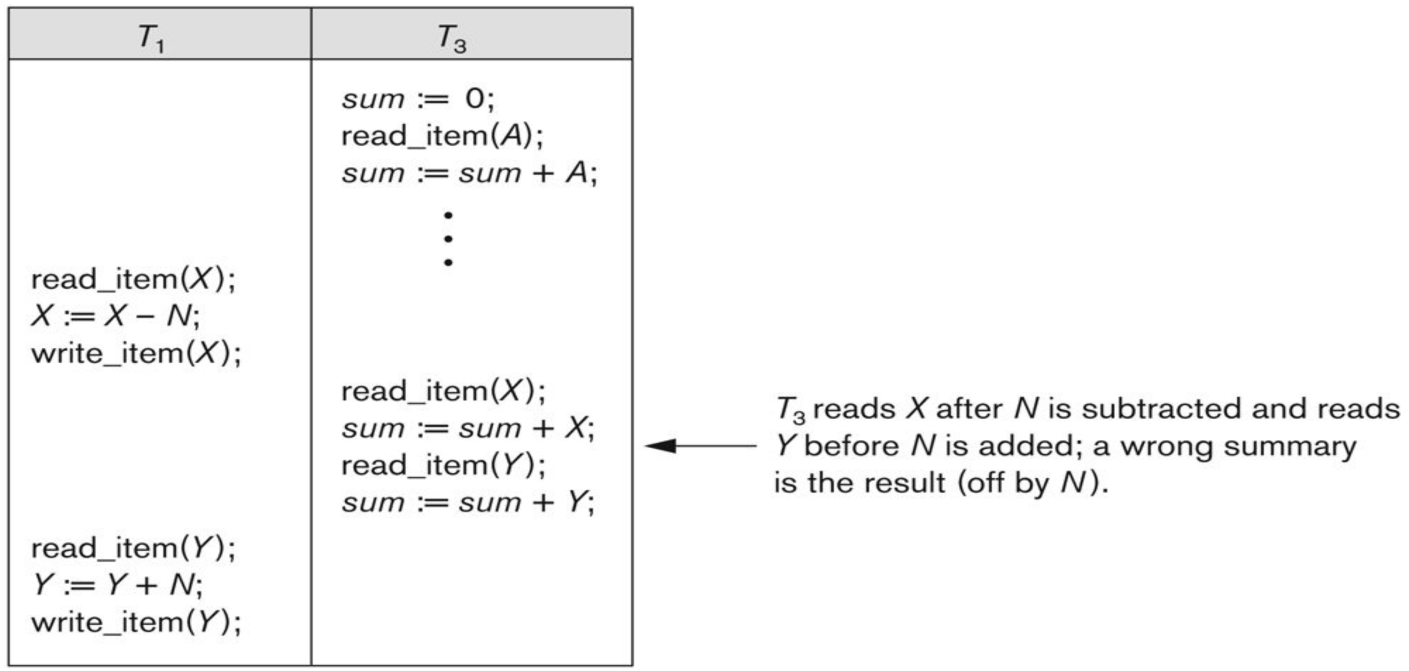
	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

Concurrency Control (Incorrect Summary Problem)

Example of **Incorrect Summary Problem** due to uncontrolled concurrent execution



T_1	T_3
	$sum := 0$; read_item(A); $sum := sum + A$; ⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Why Recovery Is Needed

What causes a Transaction to fail,

- **A computer failure (system crash):**
  A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

- **A transaction or system error:**
  Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

Why Recovery Is Needed

- **Local errors or exception conditions detected by the transaction:**
  Certain conditions necessitate cancellation of the transaction.
  - For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.
  - A programmed abort in the transaction causes it to fail.

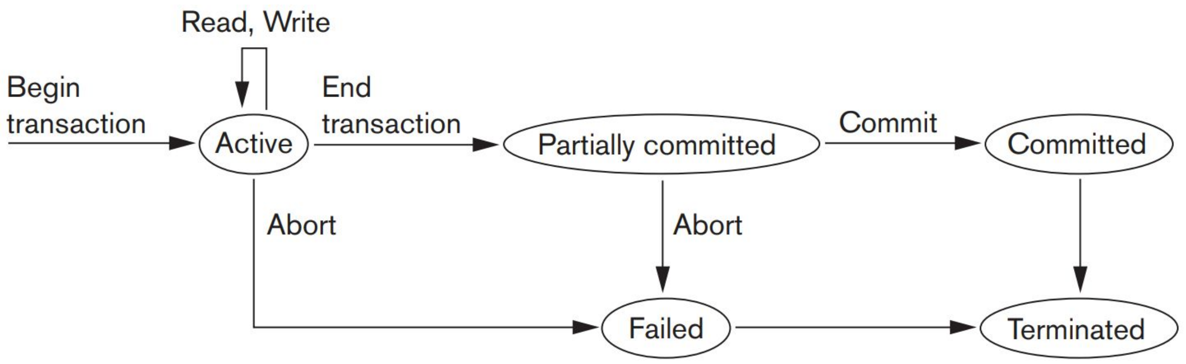
- **Concurrency control enforcement:**
  The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock

Why Recovery Is Needed

- **Disk failure:**
  Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

- **Physical problems and catastrophes:**
  This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

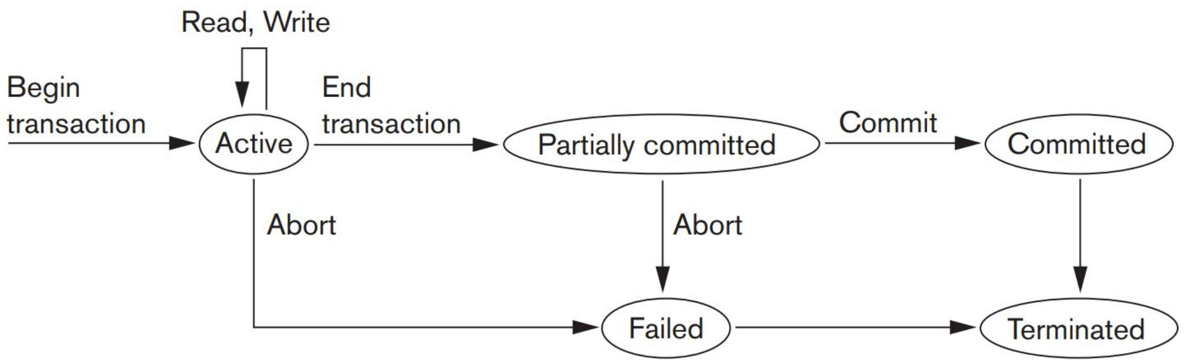
Recovery (Transaction States)



For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts. A transaction may be in one of the following states:

- A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations.
- When the transaction ends, it moves to the **partially committed state**.
- At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not. Also, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently.
- If these checks are successful, the transaction is said to have reached its commit point and enters the **committed state**.

Recovery (Transaction States)



For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts. A transaction may be in one of the following states:

- When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.
- However, a transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.
- The **terminated state** corresponds to the transaction leaving the system.
- **Failed or aborted transactions** may be restarted later—either automatically or after being resubmitted by the user—as brand new transactions.

Transaction In SQL

- A **single** SQL statement is always considered to be **atomic**. Either the statement completes execution without error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a COMMIT (commits current transaction and begins new one) or ROLLBACK(causes current transaction to abort).
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully. Implicit commit can be turned off by a database directive.
- Isolation level (serializable, repeatable read, read committed, read uncommitted) can be set at database level
- Isolation level can be changed at start of transaction

Transaction In SQL (Isolation Levels)

- **Serializable** — default, strictest level. Even if transactions are concurrent, it creates an effect that they are occurring in serial order.
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.