Transaction, Concurrency Control & Recovery

Outline

1. Introduction to Transaction
2. ACID properties
3. Concurrency Control
4. Recovery
5. Transaction in SQL

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)	BEGIN TRANSACTION; UPDATE accounts SET balance = balance - 50 WHERE account_id = 'A'; UPDATE accounts SET balance = balance + 50 WHERE account_id = 'B'; COMMIT;

- *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.*
- Read(A)/Read_Item(A) means to read data from the database while Write(A)/Write_Item(A) means to insert/delete/update data in the database.

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**
- **Consistency**
- **Isolation**
- **Durability**

ACID Properties : Atomicity

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

→ 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.

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*)

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 : Consistency

A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end, it should take the database from one consistent state to another.

→ 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.

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*).

In general, consistency requirements include,

- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints, e.g., sum of A and B is unchanged by the execution of the above transaction
- 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

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;
→ The isolation property is enforced by the concurrency control subsystem of the DBMS

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*) 4. read(*B*) 5. *B* := *B* + 50 6. write(*B*)	read(A), read(B), print(A+B)

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

ACID Properties : Durability

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

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*)

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.

Transaction Processing

Two main issues to deal with during a transaction:

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

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.

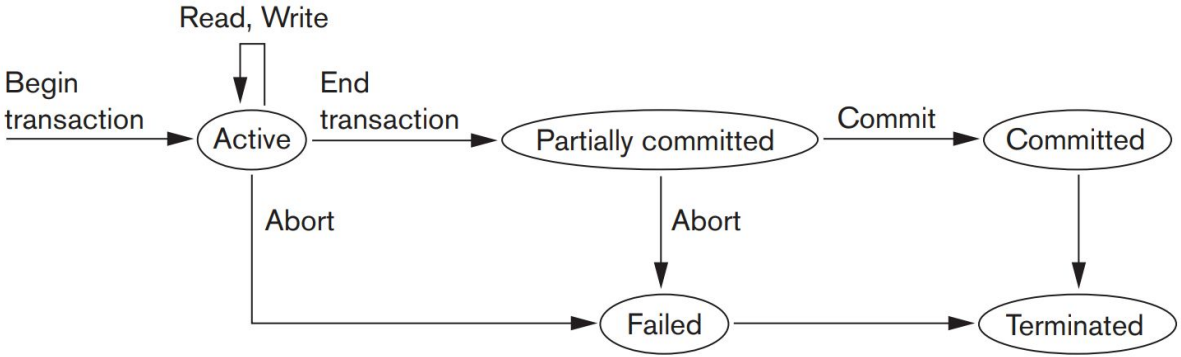
- **Concurrency control enforcement:**
  The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

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.

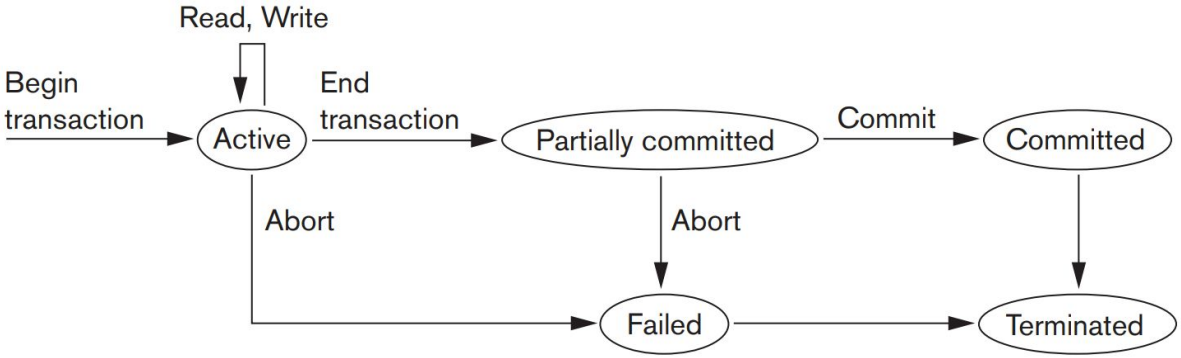
Recovery (Transaction States)



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.

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

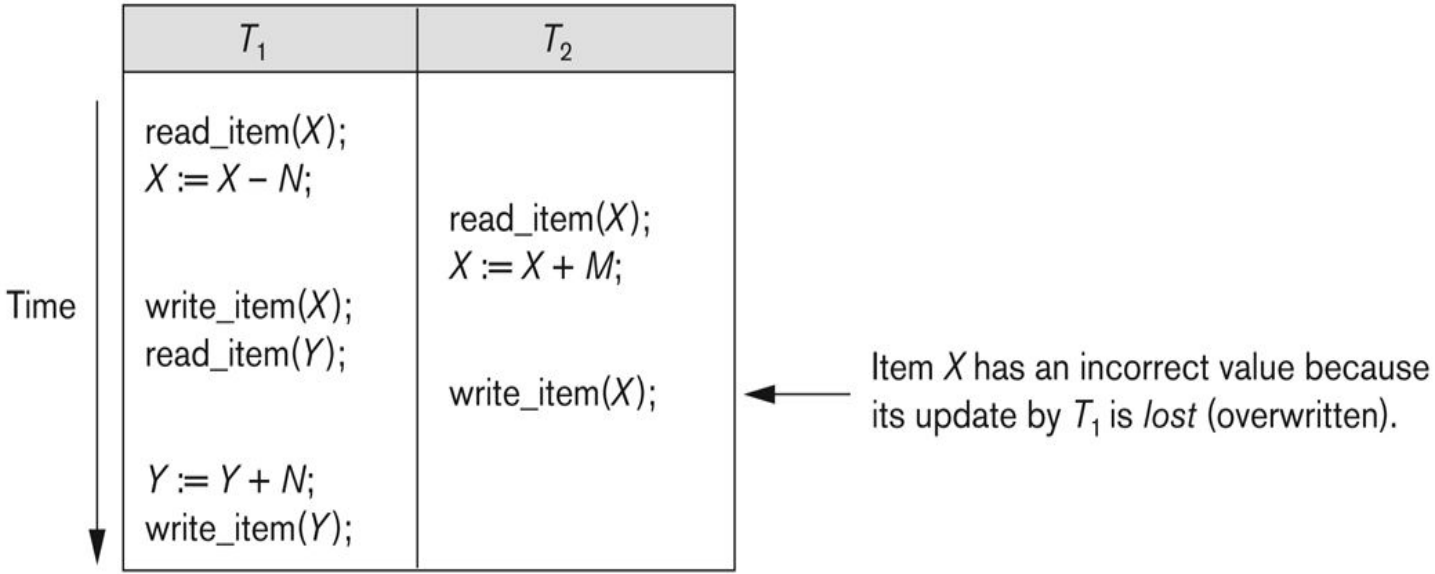
Why Concurrency Control Is Needed

If concurrency control mechanisms are not in place, then several concurrency issues/phenomena may occur due to uncontrolled concurrent transactions. Some of these issues are:

- **The Lost Update Problem**
- **The Temporary Update (or Dirty Read) Problem**
- **The Incorrect Summary (or Incorrect Analysis) Problem**
- **Non-Repeatable Read**
- **Phantom Read**

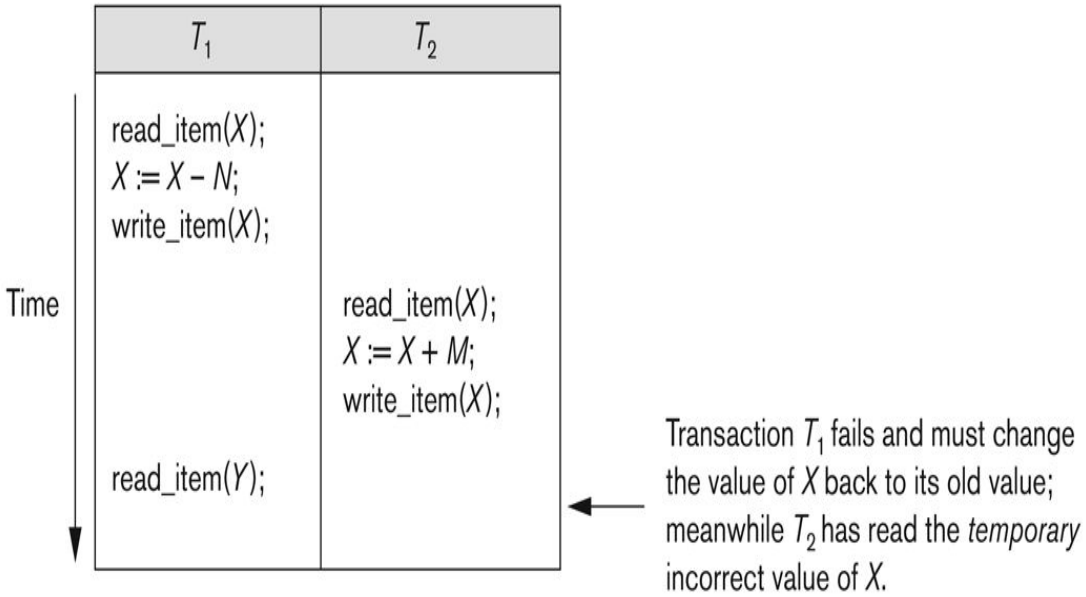
Concurrency Control Problems

Lost Update Problem: Two transactions read the same data and both update it; one update gets overwritten by the other and therefore is lost.



Concurrency Control Problems

Dirty Read Problem: A transaction reads data written by another uncommitted transaction and the uncommitted transaction gets reverted, thus invalid or non-final data is used by another transaction.



T1: UPDATE account SET balance = balance - 100 WHERE id = 1;

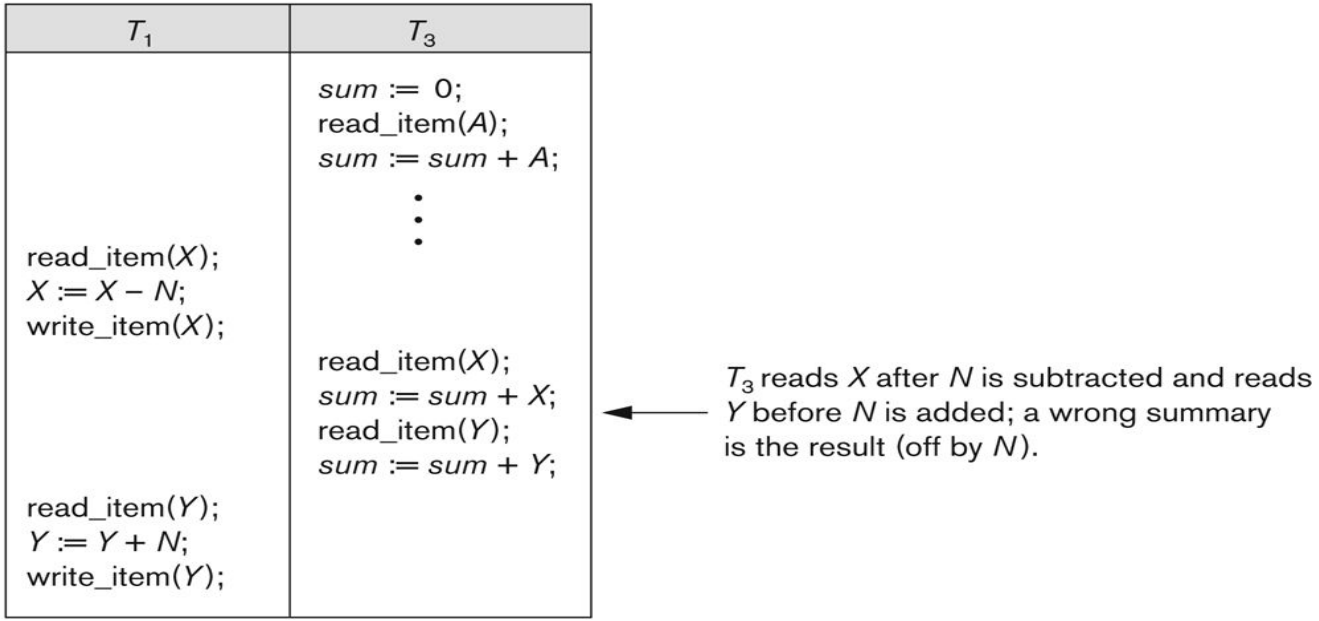
T2: SELECT balance FROM account WHERE id = 1; ← Reads uncommitted value

T1: ROLLBACK;

Now, T2 has seen data that never truly existed in the database.

Concurrency Control Problems

Incorrect Summary Problem: A transaction performs an aggregate function while other transactions are updating the data. The aggregate function may calculate some values before they are updated and others after they are updated.



Concurrency Control Problems

- **Non-Repeatable Read**: A transaction reads the same row twice and gets different values because another transaction has modified that row between the reads. This breaks consistency within the same transaction — values are unexpectedly changed.
  - Example:
    - T1: SELECT balance FROM account WHERE id = 1; → Returns $100
    - T2: UPDATE account SET balance = 200 WHERE id = 1; COMMIT;
    - T1: SELECT balance FROM account WHERE id = 1; → Now returns $200
    - T1 sees different values in the same transaction — not repeatable.
- **Phantom Read:** A transaction re-executes a query returning a set of rows that satisfy a condition, and new rows appear or disappear due to another transaction's inserts or deletes. The result set of a query changes unexpectedly during a transaction, affecting aggregates, loops, or business logic
  - Example:
    - T1: SELECT * FROM orders WHERE amount > 100; → 5 rows
    - T2: INSERT INTO orders (id, amount) VALUES (99, 150); COMMIT;
    - T1: Repeats query → Now gets 6 rows (phantom row appears)

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 levels define the degree to which a transaction must be isolated from the data modifications made by any other transaction in the database system.
- 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)

The SQL standard defines four isolation levels that can prevent the concurrency control issues to varying degrees. The three read phenomenons- Dirty Read, Non-Repeatable Read and Phantom Read are directly impacted based on the different isolation levels.

- **Serializable (Strictest)**
  - Ensures transactions appear to be executed in a serial order, even if they are concurrent.
  - Prevents all concurrency issues: dirty reads, non-repeatable reads, and phantom reads.
  - Default in some systems; strict but impacts performance.
- **Repeatable read**
  - Only committed records can be read.
  - Repeated reads of the same record return the same value.
  - Prevents dirty reads and non-repeatable reads but allows phantom reads.
  - Transactions may not be fully serializable (e.g., partial visibility of inserted records).
- **Read Committed (Default in Many Systems)**
  - Only committed records can be read.
  - Successive reads of the same record may return different committed values (due to updates by other transactions).
  - Prevents dirty reads, but non-repeatable reads and phantom reads can occur.
- **Read Uncommitted (Least Strict)**
  - Even uncommitted records can be read.
  - Allows all concurrency issues: dirty reads, non-repeatable reads, and phantom reads.
  - Fastest but provides the lowest consistency guarantee.

Transaction In SQL (Isolation Levels)

Relationship between isolation levels and concurrency control problems/phenomena

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Serializable	Prevents	Prevents	Prevents
Repeatable Read	Prevents	Prevents	Allows
Read Committed	Prevents	Allows	Allows
Read Uncommitted	Allows	Allows	Allows

Isolation levels below Serializable do not, by themselves, guarantee prevention of lost updates or incorrect summary problems. Serializable prevents all concurrency problems, while at lower levels additional techniques (locking, atomic updates, query choices/coding styles, DBMS engine features) may prevent lost updates or incorrect summary problems. Read Uncommitted does not prevent any concurrency problems.