Objectives

In this Lecture you will learn:

- What is a transaction and the transaction state
- ACID properties for maintaining database consistency.
- How and when the rollback occurs.
- The purpose of the concurrency control.

1. Introduction

A transaction is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data manipulation language (typically SQL), or programming language (e.g., C++ or Java), with embedded database accesses.

The transaction has four properties (**ACID Properties**). These are used to maintain **consistency** in a database, before and after the transaction.

1.1 Atomicity

A transaction is a single unit of operation. You either execute it entirely or do not execute it at all. There cannot be partial execution.

$$
\begin{aligned}
T_i:\ & \text{read}(A);\\
& A := A - 50;\\
& \text{write}(A);\\
& \text{read}(B);\\
& B := B + 50;\\
& \text{write}(B).
\end{aligned}
$$

Fig1. A Transaction Example

Example:

- Suppose that the values of accounts A and B are $1000 and $2000, respectively.
- Now suppose that, during the execution of transaction Ti, a failure occurs that prevents Ti from completing its execution successfully.
- Further, suppose that the failure happened after the write(A) operation but before the write(B) operation.
- In this case, the values of accounts A and B reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure.

- Because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**.

- The basic idea behind ensuring atomicity is this: The database system **keeps track** (on disk) of **the old values of any data** on which a transaction performs a write(). This information is written in a file called the **log**.

- Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system**.

1.2 Consistency

The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before the execution of the transaction, the database remains consistent after the execution of the transaction.

Mohammed Dheyaa

1.3 Durability

After successful completion of a transaction, the changes in the database should persist. Even in the case of system failures.

1.4 Isolation

A transaction should be executed in isolation from other transactions (no Locks). Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

1.5 Transaction states

- a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Once the changes caused by an aborted transaction have been **undone**, we say that the transaction has been **rolled back**. A transaction that completes its execution successfully is said to be **committed.**

- A transaction must be in one of the following states:

- **Active**, is 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 has been restored to its state prior to the start of the transaction.
- **Committed**, after successful completion.

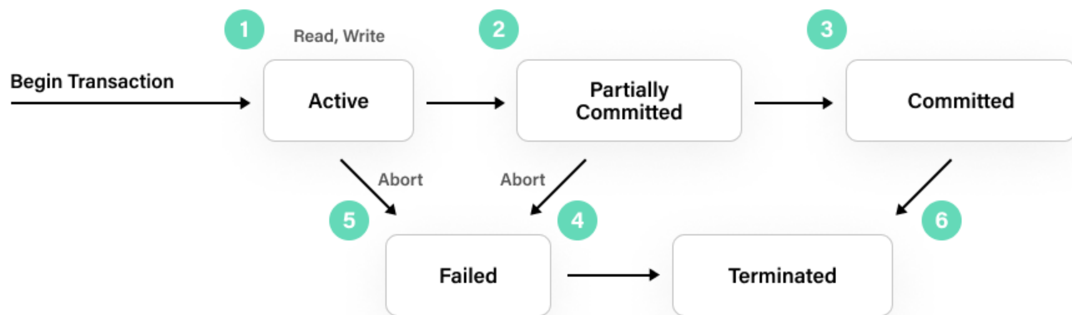

Fig2. A Transaction States.

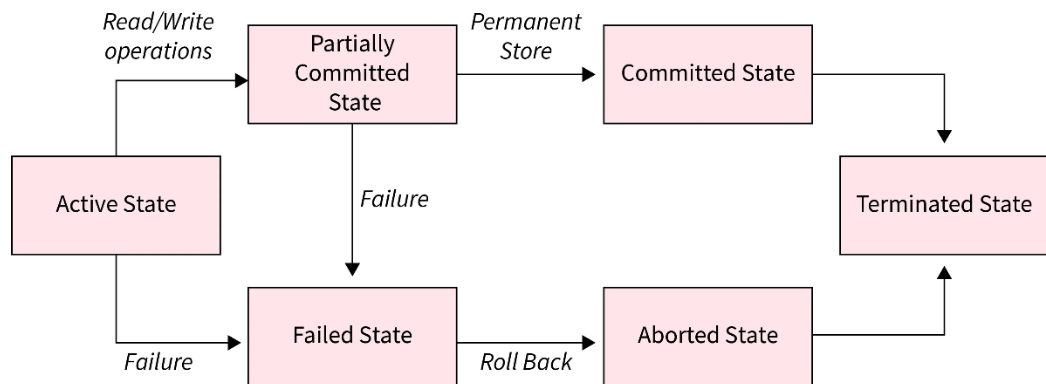

Fig3. A Transaction States with Roll Back Mechanism.

1.6 Transaction Isolating

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with the consistency of the data, as we saw earlier. Concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has been completed. However, there are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization:** The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increases; in other words, the processor and disk spend less time idle, or not performing any useful work.

- **Reduced waiting time.**

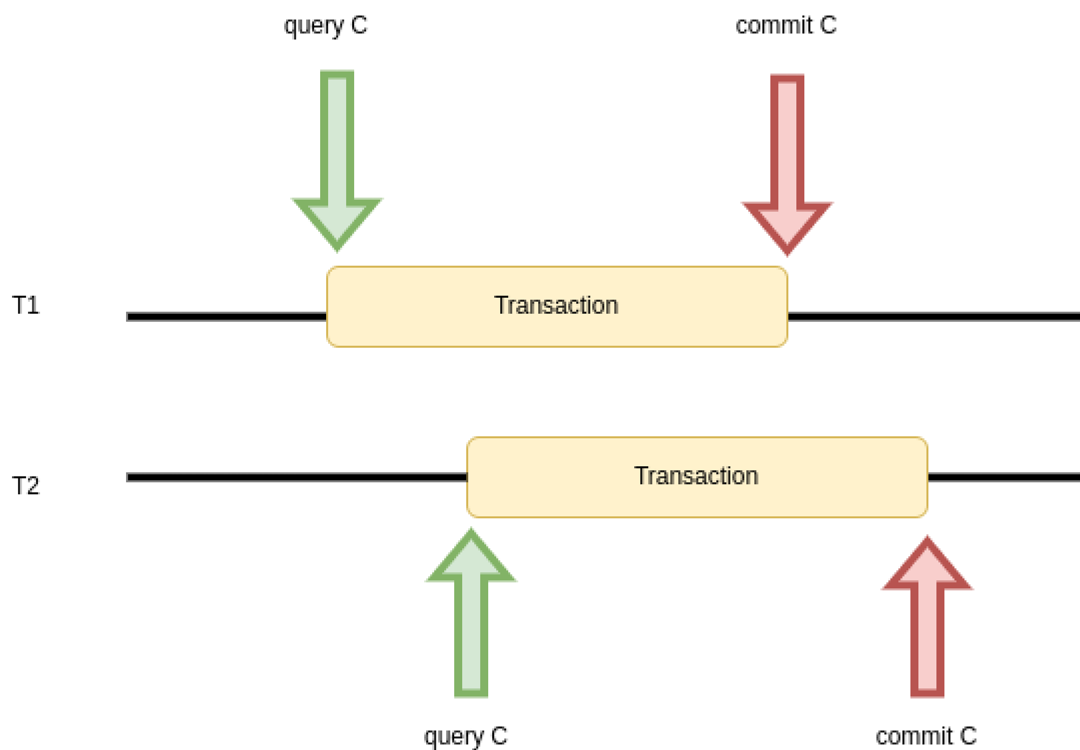

Fig4. Concurrent Transactions.

2. Concurrency Control

Concurrency, as defined by Merriam-Webster, is two or more things capable of "operating or occurring at the same time."

In the context of software, Concurrency Control is the process of managing simultaneous operations on the database without having them interfere with one another.

- A major objective in developing a database is to enable **many users to access shared data concurrently**.
- Concurrent access is relatively easy if **all users are only reading data**, as there is no way that they can interfere with one another
- When two or more users are accessing the database **simultaneously** and at least one is **updating** data, there may be **interference** that can result in inconsistencies.

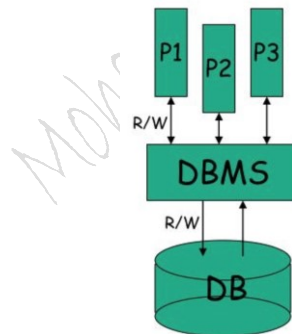

Fig5. Concurrent Transactions.

- **The Concurrency Control** objective is similar to the objective of multi-user computer systems, which allow two or more programs (or transactions) to execute at the same time.
- The operations of the two transactions are **interleaved** to achieve concurrent execution. In addition, throughput—the amount of work that is accomplished in a given time interval—is improved as the CPU executes other transactions instead of being in an idle state.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Fig. 6. Schedule 1—a serial schedule followed by T_1.

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

Fig. 7 Schedule 2—a serial schedule in which T_2 is T_1 is followed by T_2.

- These schedules are serial. Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.
- When the database system executes several transactions concurrently, the corresponding **schedule no longer needs to be serial**.
- If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Fig. 8 Schedule 3—a concurrent schedule equivalent to schedule 1.

- It is the job of the database system to ensure that any schedule that is executed will leave the database in a **consistent state**.

- The concurrency control carries out this task.

- We can ensure consistency of the database under concurrent execution by making sure that any **schedule** that is executed has the **same** effect as a **schedule** that could have occurred **without any concurrent execution**.

- That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable schedules**.

We examine **three** examples of potential problems caused by concurrency (Concurrency phenomena, Concurrency conflicts):

- The lost update problem.
- The uncommitted dependency problem.
- The inconsistent analysis problem.

Mohammed Dheyaa

2.1 Lost Update

The lost update problem occurs when **multiple concurrent transactions** try to **read** and **update the same data**. Let us understand this with the help of examples:

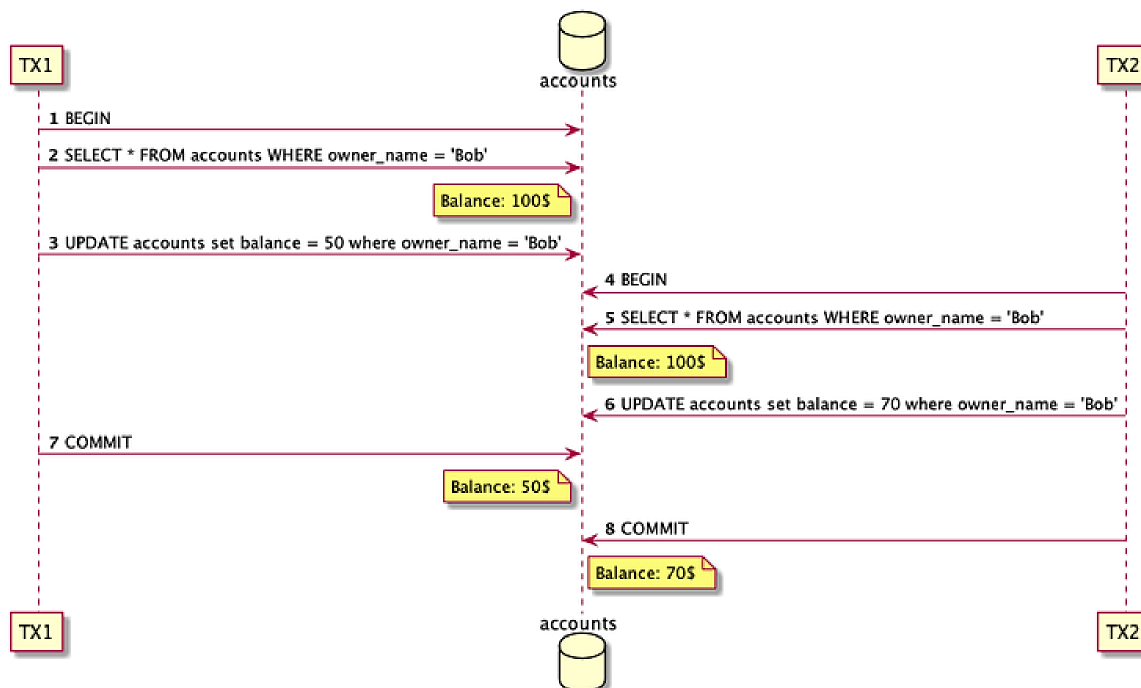

Fig. 9 The lost update problem.

In this example, we have two concurrent transactions trying to withdraw 50$ and 30$from Bob's account. After both transitions have finished we expect to see 20$ on the account balance, but since the second transaction reads only the committed data, it is unaware of the concurrent operation and behaves as the first transaction never happened. As a result, the second transaction overwrote the first update, and our system suffered a 50$ loss. This was an example of the **Lost Update problem**.

Example 2: The loss of T2's update is avoided by preventing T1 from reading the value of **bal**$_x$ until after T2's update has been completed.

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Fig. 10 The lost update problem.

2.2 The uncommitted dependency (or dirty read) problem

A dirty read happens when a transaction is allowed to read the **uncommitted changes** of some other concurrent transaction.

Taking a business decision on a value that has not been committed is risky because uncommitted changes might get rolled back.

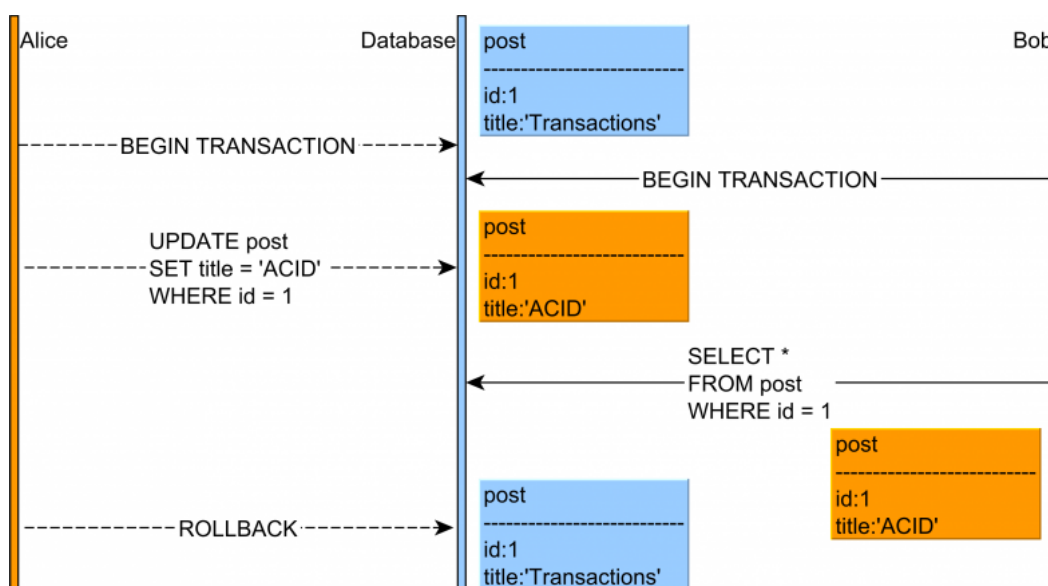

Fig. 11. The dirty read problem.

In the diagram above, the flow of statements goes like this:

1. Alice and Bob started two database transactions.
2. Alice modifies the title of a given post record.
3. Bob reads the uncommitted post record.
4. If Alice commits her transaction, everything is fine. But if Alice **rolls back,** then Bob will see a record version that no longer exists in the database transaction log.

Rolls back an explicit or implicit transaction to the beginning of the transaction, or to a savepoint inside the transaction.

You can use **ROLLBACK TRANSACTION** to erase all data modifications made from the start of the transaction or to a savepoint. It also frees resources held by the transaction.

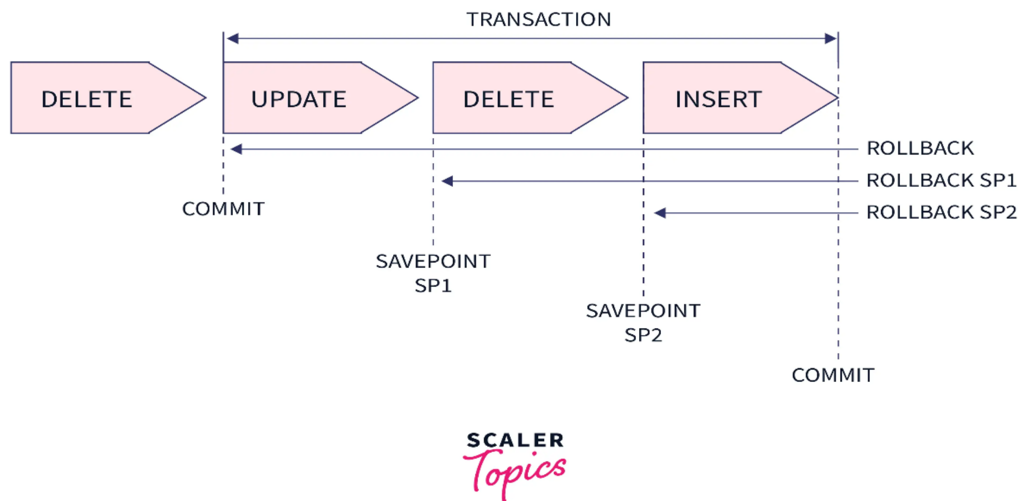

Fig.12 . A Transaction Rollback.

Example :

The uncommitted dependency problem occurs when one transaction is allowed to see the intermediate results of another transaction before it has been committed.

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

Fig. 13 . The uncommitted dependency problem.

2.3 The inconsistent analysis problem

The problem of inconsistent analysis occurs when a **transaction reads several values** from the database but a **second transaction updates some of them during the execution of the first**.

For example, a transaction that is summarizing data in a database (for example, totaling balances) will obtain inaccurate results if, while it is executing, other transactions are updating the database.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Fig.14. inconsistent analysis.