Transaction Management

- Lecture outline

- Overview
- Definition and motivation for transactions
- The ACID properties of transactions
- Transaction states
- Concurrent Executions and Schedules
- Conflict Serializability
- Testing for Serializability
- View Serializability
- Transaction Definition in SQL

- Overview

- The database system must ensure that the data stored in the database is always ***consistent***.
- There are several possible types of ***failures*** that may cause the data to become inconsistent.
- A ***transaction*** is an ***atomic*** program that executes on the database and preserves the consistency of the database.
- The input to a transaction is a consistent database, AND the output of the transaction must also be a consistent database.
- A transaction must execute completely or not at all.

-- Motivating Example

- Consider a person who wants to transfer $50 from a savings account with balance $1000 to a checking account with current balance = $250.

  1. At the ATM, the person starts the process by telling the bank to remove $50 from the savings account.
  2. The $50 is removed from the savings account by the bank.
  3. Before the customer can tell the ATM to deposit the $50 in the checking account, the ATM “crashes.”

- Where has the $50 gone?
- It is lost if the ATM did not support transactions!
- The customer wanted the withdraw and deposit to both happen in one step, or neither action to happen.

- Transaction Definition

- A ***transaction*** is an ***atomic*** program that executes on the database and preserves the consistency of the database.
- The basic assumption is that when a transaction starts executing the database is consistent, and when it finishes executing the database is still in a consistent state.
- Do not consider malicious or incorrect transactions.
- This assumption is called **The Correctness Principle**.
- Note that the database may be inconsistent during transaction execution.
- For the bank example, the $50 is removed from the savings account and is not yet in the checking account at some point in time.

Consistent Definition

- A database is ***consistent*** if the data satisfies all constraints
- specified in the database schema. A ***consistent database*** is said to be in a ***consistent state***.
- A ***constraint*** is a predicate (rule) that the data must satisfy.
- Examples:
  - *StudentID* is a key of relation *Student*.
  - *StudentID* → *Name* holds in *Student*.
  - No student may have more than one major.
  - The field *Major* can only have one of the 4 values: {“BA”,”BS”,”CS”,”ME”}.
  - The field *Year* must be between 1 and 4.

Consistency and Constraints

- Note that constraints are logical rules that may not capture a complete view of all data integrity “issues”.

  1. Database constraints do not typically capture transaction constraints. These are data integrity issues built into transactions themselves such as:
     - The *Year* field is updated every September by increasing its value by 1, only if the degree requirements are met.
  2. Since a database only models the real-world, the data it contains and the associated constraints may not reflect the total picture. For example:
     - The *Year* field does not adequately reflect how many years the student has been attending university, only the year they are in with respect to their program degree.

Consistency Issues

- There are two major types of challenges in preserving database consistency:

1. The database system must handle ***failures*** of various kinds such as hardware failures and system crashes.

2. The database system must support ***concurrent execution*** of multiple transactions and guarantee that this concurrency does not lead to inconsistency.

ACID Properties

- To preserve integrity, transactions have the following properties:
  - ***Atomicity*** **-** Either all operations of the transaction are properly reflected in the database or none are.
  - ***Consistency*** **-** Execution of a transaction in isolation preserves the consistency of the database.
  - ***Isolation*** **-** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
    - Intermediate transaction results must be hidden from other concurrently executing transactions. That is, for every pair of transactions *Ti* and *Tj,* it appears to *Ti* that either *Tj,* finished execution before *Ti* started, or *Tj* started execution after *Ti* finished.
  - ***Durability*** **-** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction Operations

- Since a transaction is a general program, there are an enormous number of potential operations that a transaction can perform.
- However, there are only two really important operations:
  - read(A,t) (or read(A) when *t* is not important)
    - Read database element *A* into local variable *t*.
  - write(A,t) (or write(A) when *t* is not important)
    - Write the value of local variable *t* to the database element *A*.
- For most of the discussion, we will assume that the buffer manager insures that database element is in memory. We could make the memory management more explicit by using:
  - input(A)
    - Read database element *A* into local memory buffer.
  - output(A)
    - Copy the block containing *A* to disk.

Fund Transfer Transaction Example ...

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

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

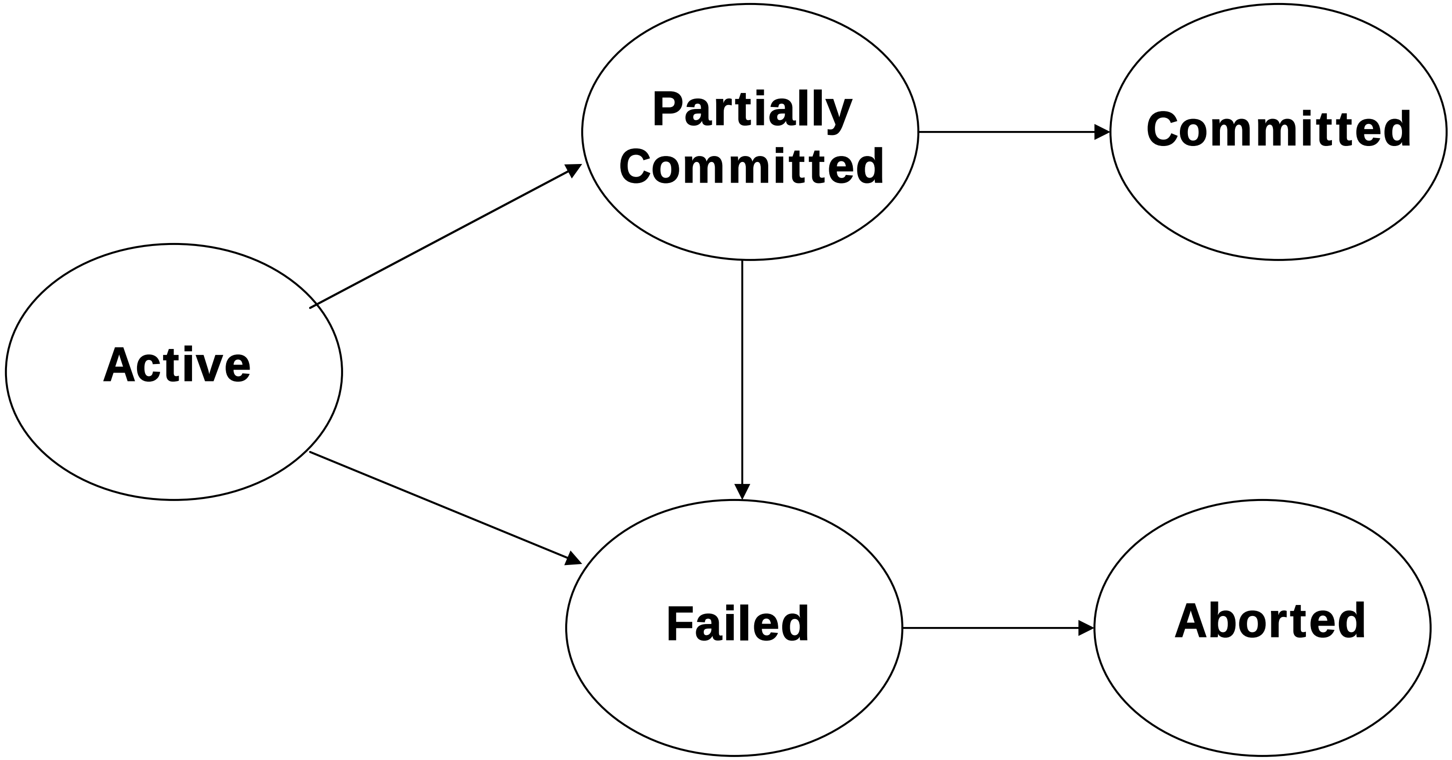
... Fund Transfer Transaction Example

- ***Atomicity requirement*** – If the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, or inconsistency will result.
- ***Consistency requirement*** – The sum of *A* and *B* is unchanged by the execution of the transaction.
- ***Isolation requirement*** – If between steps 3 and 6, another transaction accesses the partially updated database, it will see an inconsistent database (*A + B* is less than it should be). Can be ensured trivially by running transactions *serially,* that is one after the other. However, executing multiple transactions concurrently has significant benefits.
- ***Durability requirement*** – Once the user has been notified that the transaction has completed (i.e., the $50 transfer occurred), the updates by the transaction must persist despite failures.

Transaction States

- During its execution, a transaction can be in many states:
  - **Active** - is the initial state. The transaction stays in this state while it is executing.
  - **Partially committed -** A transaction is partially committed after its final statement has been executed.
  - **Failed -** A transaction enters the failed state after the discovery that normal execution can no longer proceed.
  - **Aborted -** A transaction is aborted after it has been rolled back and the database restored to its prior state before the transaction. There are two options after abort:
    - restart the transaction – only if no internal logical error
    - kill the transaction - problem with transaction itself
  - **Committed -** Commit state occurs after successful completion.
    - May also consider **terminated** as a transaction state.

Transaction State Diagram



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - Increased processor and disk utilization, leading to better transaction ***throughput****:* one transaction can be using the CPU while another is reading from or writing to the disk.
  - Reduced ***average response time*** for transactions as short transactions need not wait behind long ones.
- ***Concurrency control schemes*** are mechanisms to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.
  - We will study concurrency control schemes after examining the notion of correctness of concurrent executions.

END