Database Management Systems (DBMS)

Lec 24: Transaction Processing, Concurrency Control, and Recovery

Ramesh K. Jallu
IIIT Raichur

Date: 08/06/21

Transactions

- A ***transaction*** is a collection of operations that form a single logical unit of database processing
    - E.x., transfer of money from one account to another is a transaction, booking a reservation, online purchase, etc.

- A transaction is typically implemented by a computer program that includes database commands such as retrievals, insertions, deletions, and updates

- The operations in a transaction can either be embedded within an application program or they can be specified interactively via SQL

Transactions (Contd.)

- We focus on the basic concepts and theory that are needed to ensure the correct executions of transactions

- The desirable properties of transactions:
  - ***Atomicity***, ***Consistency***, ***Isolation***, and ***Durability***

ACID properties

- **Atomicity**: Either all operations of the transaction are reflected properly in the database, or none are

- **Consistency**: The integrity constraints must be maintained so that the database is consistent before and after the transaction
  - Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction

- **Isolation**: Each transaction is unaware of other transactions executing concurrently in the system

- **Durability**: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Transactions (Contd.)

- A transaction is delimited by statements (or function calls) of the form ***begin transaction*** and ***end transaction***

- The transaction consists of ***all operations*** executed between the begin transaction and end transaction

- If a transaction *do not update the database but only retrieve data*, the transaction is called a ***read-only transaction***; otherwise it is known as a ***read-write transaction***

Transaction model

- We consider a simple database language that focuses on when data are moved from disk to main memory and from main memory to disk

- The only actual operations on the data are restricted in our simple language to arithmetic operations

- A **database** is basically represented as a collection of *named data items* and the size of a data item is called its ***granularity***

- The transaction processing concepts we discuss are independent of the data item granularity and apply to data items in general

- The data items in our simplified model have *unique* name and contain a *single* data value

The two basic operations

- Let us consider a simple bank application consisting of several accounts and a set of transactions that access and update those accounts

- Transactions access data using two operations:

1. ***read_item(X)*, or simply *read(X)*:**
   - Reads a database item named ***X*** into a program variable, also called ***X***
     - ***I.e,*** $X \leftarrow$ **read(*X*)**

2. ***write_item(X)***, or simply ***write(X):***
   - Writes the value of program variable ***X*** into the database item named ***X***
     - ***I.e,*** $X \leftarrow$ write(***X***)

Data transfer from disk to main memory

- The basic unit of data transfer from disk to main memory is one disk page (disk block)
- The DB system maintains a *database cache*, a number of **data buffers** in main memory
- Each buffer typically holds the contents of one database disk block
- When a buffer overflow occurs, buffer replacement polycies are used (such as LRU, FIFO, etc.)
- If the chosen buffer has been modified, it must be written back to disk before it is reused

The execution of read_item(*X*)

1. Find the address of the disk block that contains item *X*

2. 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

3. Copy item *X* from the buffer to the program variable named *X*

The execution of write_item(*X*)

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

Example

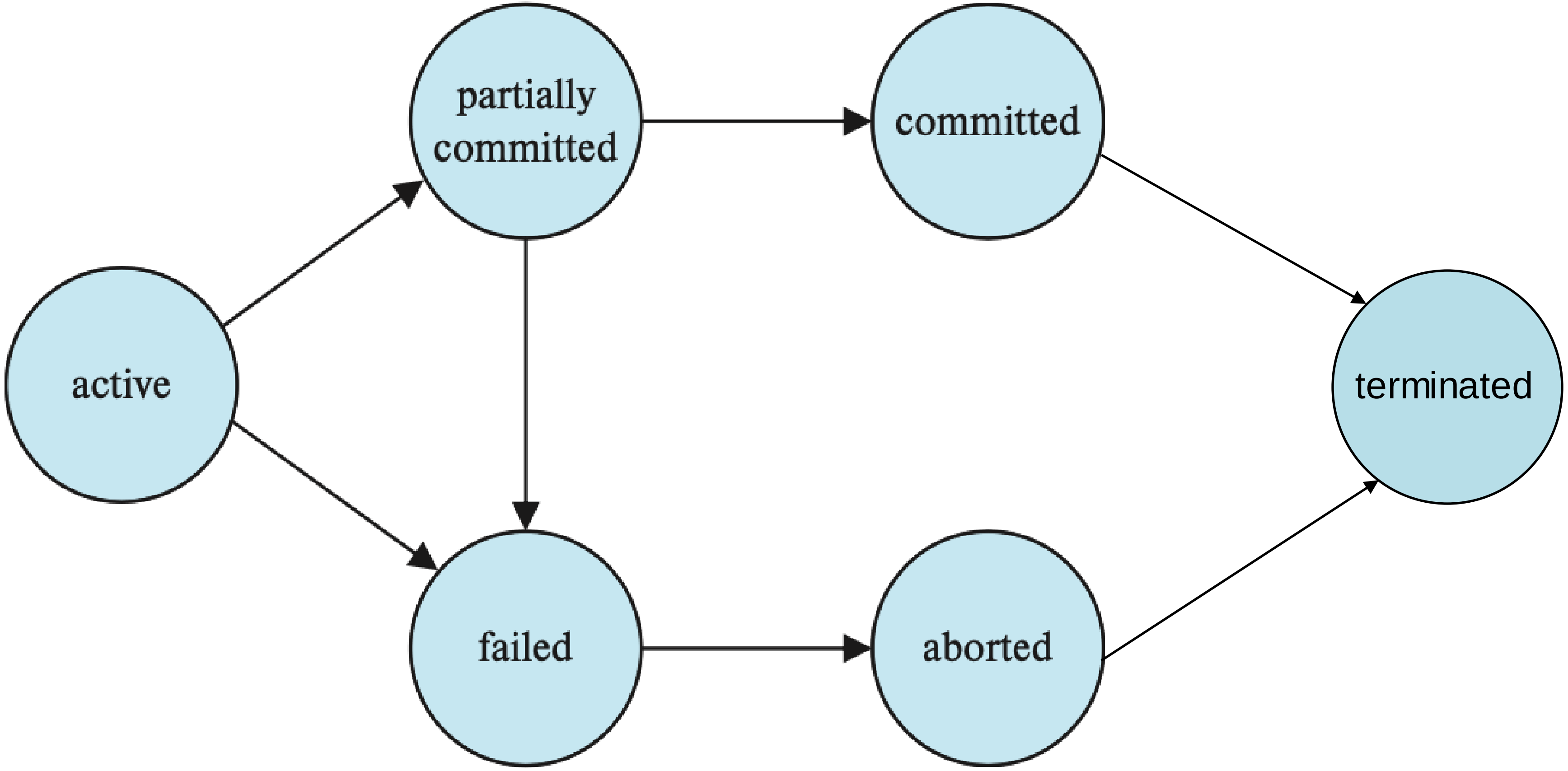
- Let T_i be a transaction that transfers Rs. 50 from account A to account B

- This transaction can be defined as:

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

1. Consistency
2. Atomicity
3. Durable
4. Isolation

Different states in transactions



Schedules

- The execution sequences are called ***schedules***
- They represent the chronological order in which instructions are executed in the system
- A schedule is a ***serial schedule*** if each transaction starts only after the previous one has completed

  All serial schedules leave the system in consistent state
- For a set of *n* transactions, there exist *n*! different valid serial schedules
- Transactions can either be executed serially or concurrently

Concurrent transactions

- 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 2nd, then 1st, and so on

- With multiple transactions, the CPU time is shared among all the transactions

- Several execution sequences are possible, since the various instructions from multiple transactions are interleaved

Concurrent transactions (Contd.)

- Two good reasons for allowing concurrency
  - Improved throughput and resource utilization
  - Reduced waiting time

- Allowing multiple transactions to update data concurrently causes several complications with consistency of the data

Not all concurrent schedules result in a consistent state

- When several transactions run concurrently, the isolation property may be violated

- The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency in the system

Example: Serial execution

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

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

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	

Example: Concurrent execution

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

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

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

Serializability

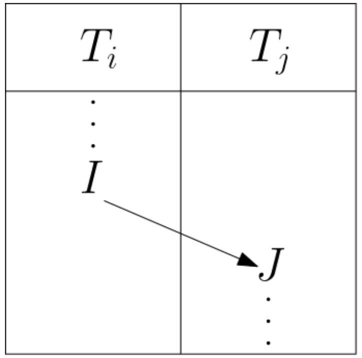
- We can ensure the consistency under concurrent execution by making sure that the concurrent execution has the same effect as a serial schedule
  - That is, the schedule should be equivalent to a serial schedule
- Such schedules are called ***serializable*** schedules
- How to determine a schedule is serializable?

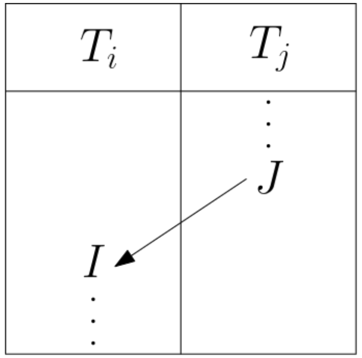
Conflict serializability

- Let us consider a schedule ***S*** in which there are two consecutive instructions, *I* and *J*, of transactions T_i and T_j, respectively ($i \neq j$)

- If *I* and *J* refer to different data items, then we can swap *I* and *J* without affecting the results of any instruction in the schedule

- If *I* and *J* refer to the same data item *X*, then the order of the two steps may matter: there are four cases that we need to consider
  1. *I = read(X), J = read(X);*
  2. *I = read(X), J = write(X);*
  3. *I = write(X), J = read(X);*
  4. *I = write(X), J = write(X);*





Conflict serializability (Contd.)

- We say that I and J ***conflict*** if they are operations by different transactions on the same data item, and at least one of these instructions is a ***write*** operation
- If I and J do not conflict, then we can swap the order of I and J to produce a new schedule $\boldsymbol{S'}$
- $\boldsymbol{S}$ is equivalent to $\boldsymbol{S'}$, since all instructions appear in the same order in both schedules except for I and J, whose order does not matter
- If a schedule $\boldsymbol{S}$ can be transformed into a schedule $\boldsymbol{S'}$ by a series of swaps of nonconflicting instructions, we say that $\boldsymbol{S}$ and $\boldsymbol{S'}$ are ***conflict equivalent***

Conflict serializability (Contd.)

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Conflict serializability (Contd.)

- Are serial schedules conflict equivalent to each other?
  - I.e., if S and T are two serial schedules, can we obtain S from T (or vice versa) by a series of swapings?

- We say that a schedule ***S*** is ***conflict serializable*** if it is conflict equivalent to a serial schedule

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Thank you!