Week 14, Lec 27

Transaction and Concurrency Control

Part 1

Contents

- **Transaction control**
- **Data Concurrency and Consistency in a Multiuser Environment**
- **Locking**

A Banking Transaction

Transaction Begins

```
UPDATE savings_accounts
     SET balance = balance - 500
     WHERE account = 3209;
```

Decrement Savings Account

```
UPDATE checking_accounts
     SET balance = balance + 500
     WHERE account = 3208;
```

Increment Checking Account

```
INSERT INTO journal VALUES
     (journal_seq.NEXTVAL, '1B'
     3209, 3208, 500);
```

Record in Transaction Journal

```
COMMIT WORK;
```

End Transaction

Transaction Ends

Transactions - Rationale

- Consider two clients booking airline tickets
- There are 2 seats left on a flight
- Client A wants 2 seats:
  - time 12:02 makes initial request
  - 12:06 confirms purchase through booking form
  - 12:08 authorises credit card payment
- Client B wants 2 seats:
  - time 12:03 makes initial request
  - 12:05 confirms purchase through booking form
  - 12:09 authorises credit card payment
- Situation needs careful control

Some Possibilities

- Clients A and B are both told 2 seats are free in initial enquiries
- B confirms purchase before A
  - But A may still proceed
- A attempts credit card debit first
  - If successful A secures tickets at 12:08
- B then attempts credit card debit
  - If successful B secures tickets at 12:09
    - potentially over-writing A's tickets
    - A has paid for tickets no longer his/hers

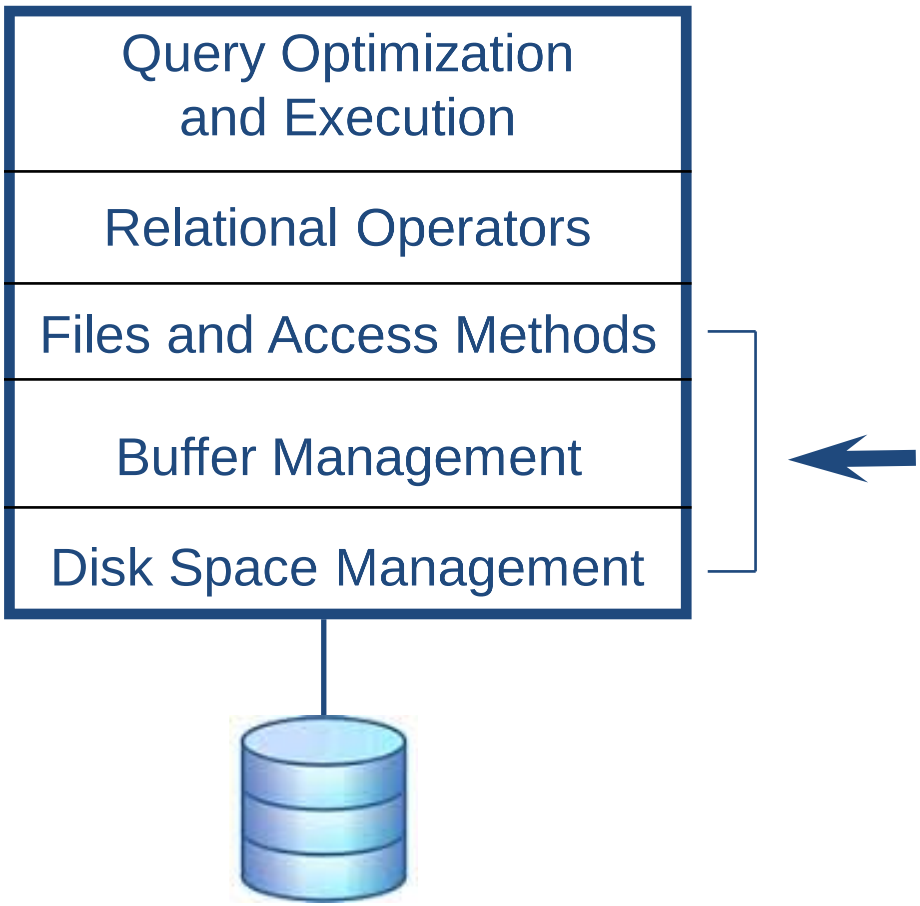
Requirements 1

- When client A beats B in the initial enquiry:
  - they should form a queue (serialisability)
  - B must wait for A to finish
- Different kinds of finish for A:
  - successful
    - completes booking form
    - makes credit card debit
    - store results (commit)
      - number of seats available is now zero
    - write transaction log and finish
    - B cannot proceed with purchase as no tickets left

Requirements 2

  - unsuccessful
    - may not complete booking form
    - may not have funds on credit card
    - undo any database changes (rollback) and finish
      - number of seats available is still 2
    - B can now proceed to attempt to purchase the 2 tickets left
- Techniques required to emulate business practice

Structure of a DBMS



These layers must consider concurrency
control and recovery
(Transaction, Lock, Recovery Managers)

Transactions and Concurrent Execution

- Transaction - DBMS's abstract view of a user program (or activity):
  - **A sequence of reads and writes of database objects.**
  - Unit of work that must commit or abort as an atomic unit
- Transaction Manager controls the execution of transactions.
- User's program logic is invisible to DBMS!
  - Arbitrary computation possible on data fetched from the DB
  - The DBMS only sees data read/written from/to the DB.

- Challenge: provide atomic transactions to concurrent users!
  - Given only the read/write interface.

Concurrency: Why bother?

- The *latency* argument
  - Latency
    - Average response time
    - Average time taken to complete a transaction
- The *throughput* argument
  - System throughput:
    - Number of transactions executed per time unit
- Both are critical!

Example of a Fund Transfer

- **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.

ACID Properties

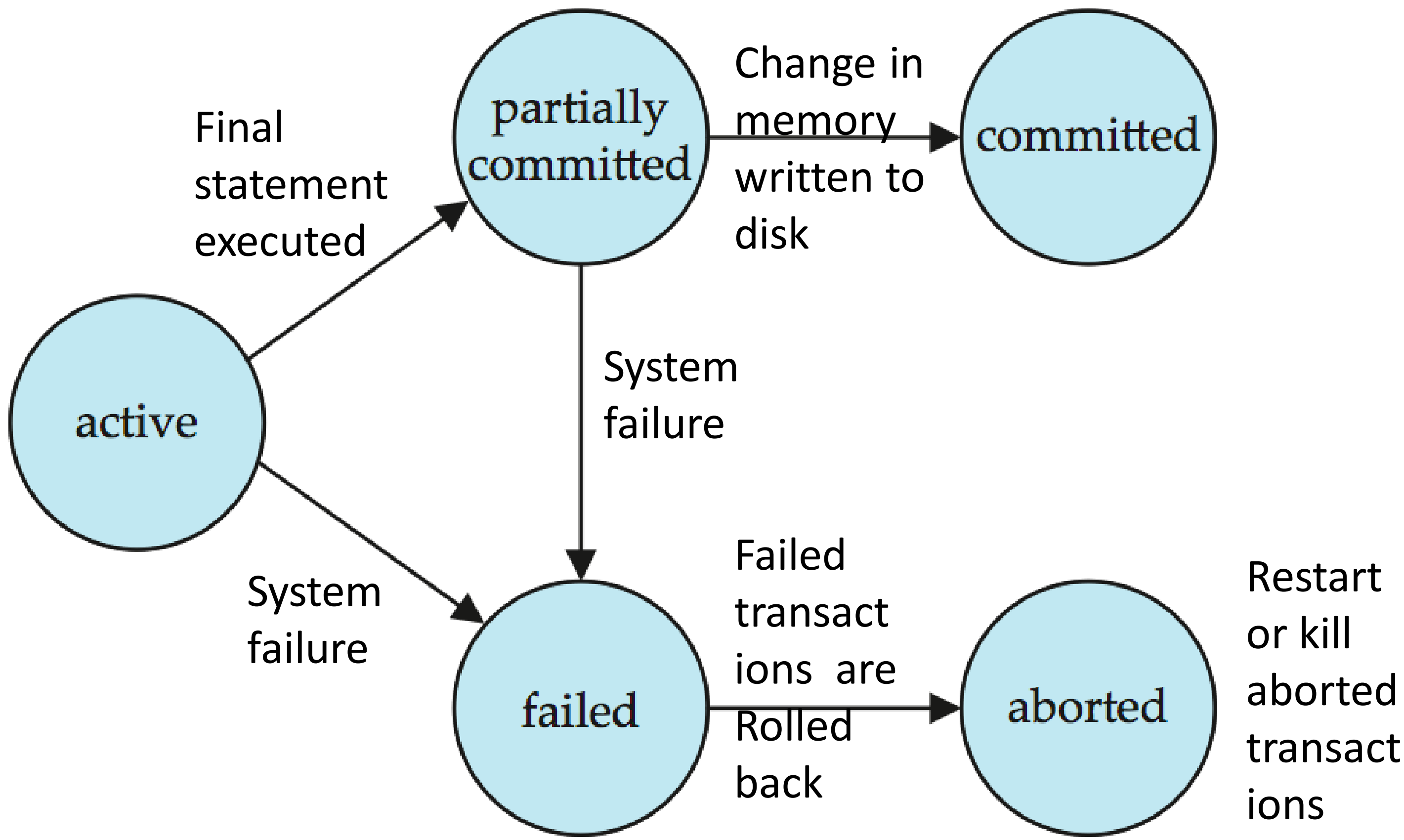
A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

- **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 executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – 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 restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction
    - can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



A.C.I.D.

Atomicity and Durability

- A transaction ends in one of two ways:
  - *commit* after completing all its actions
    - "commit" is a contract with the caller of the DB
  - *abort* (or be aborted by the DBMS) after executing some actions.
    - Or *system crash* while the xact is in progress; treat as abort.
- Two important properties for a transaction:
  - *Atomicity* : Either execute all its actions, or none of them
  - *Durability* : The effects of a committed transaction must survive failures.
- DBMS ensures the above by *logging* all actions:
  - *Undo* the actions of aborted/failed transactions.
  - *Redo* actions of committed transactions not yet propagated to disk when system crashes.

SQL Transaction commands

- DBMS does not have an built-in way of knowing which commands are grouped to form a single logical transaction.
- Some commands, e.g. **COMMIT** and **ROLLBACK** can provide boundaries of transaction.
- **Commit**
  - saves current database state
  - releases resources, locks & savepoints held
  - equivalent to Save and Exit in MS Word
- **Rollback**
  - returns database state to that at start of transaction
  - releases resources, locks & savepoints held
  - equivalent to dismiss/ do not save changes in MS Word
- By default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off
    - E.g. in SQLPLUS, set autocommit off

Transactions in SQL

- A transaction is a logical unit of work on a database.
- A group of related operations that
  - typically comprises a collection of individual actions
    - e.g. in SQL INSERT, UPDATE, DELETE, SELECT
  - must be performed successfully
    - before any changes to the database are finalised.
- Variable size:
  - entire run on SQL*Plus
    - e.g. spend 2 hours inserting data
  - single command in SQL*Plus
    - e.g. one insert command
  - one execution of a procedure
    - e.g. one run of add_patient

Database Transaction

A database transaction consists of one of the following:

- **DML statements which constitute one consistent change to the data**
- **One DDL statement**
- **One DCL statement**

Oracle Transaction Types

Type	Description
Data manipulation language (DML)	Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work
Data definition language (DDL)	Consists of only one DDL statement
Data control language (DCL)	Consists of only one DCL statement (GRANT, REVOKE)

Transaction boundaries

A transaction begins with the first executable SQL statement.

A transaction ends with one of the following events:

- **A COMMIT or ROLLBACK statement is issued**
- **A DDL or DCL statement executes (automatic commit)**
- **The user exits *i*SQL*Plus**
- **The system crashes**

Advantages of COMMIT and ROLLBACK

With COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations

Concurrent Executions

- 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
  - Control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- Goal: A + B is "preserved".
- A serial schedule in which T_1 is followed by T_2 :

A = 100
B = 10

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

Schedule 2

- A serial schedule where T_2 is followed by T_1

A = 100
B = 10

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	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

A = 100
B = 10

	T_1	T_2	
A = 100 A = 50 Write A = 50	read (A) $A := A - 50$ write (A)		
		read (A) $temp := A * 0.1$ $A := A - temp$ write (A)	A = 50 temp = 5 A = 45 **Write A = 45**
B = 10 B = 60 Write B = 60	read (B) $B := B + 50$ write (B) commit		
		read (B) $B := B + temp$ write (B) commit	B = 60 B = 65 **Write B = 65**

A + B = 110

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following **concurrent** schedule ***does not* preserve** the value of (A + B).

A = 100
B = 10

	T_1	T_2	
A = 100	read (A)		
A = 50	$A := A - 50$		
		read (A)	A = 100
		$temp := A * 0.1$	temp = 10
		$A := A - temp$	A = 90
		write (A)	Write A = 90
		read (B)	B = 10
Write A = 50	write (A)		
B = 10	read (B)		
B = 60	$B := B + 50$		
Write B = 60	write (B)		
	commit		
		$B := B + temp$	B = 10 + 10 = 20
		write (B)	**Write B = 20**
		commit	

A + B = 70

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions I_i and I_j of 2 transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into another schedule *S´* by a series of swaps of *non-conflicting instructions*, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

Transforming details were discussed on board in class. (to be continued)

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

Schedule 3

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

Schedule 6