DATABASE MANAGEMENT SYSTEM

Subash Manandhar

Chapter 5 : Relational Database Design

- **Assertions and Triggers**
  - **Assertions:**
  - Are general purpose checks that allow the enforcement of any condition over the entire database.
  - When the assertion is made, the system tests its validity and tests it again on every update that may violate the assertion.
  - This testing may introduce a significant amount of overhead, hence assertion should be used with great care.

Chapter 5 : Database Constraints and Relational Database Design

- **Assertions:**
- CREATE ASSERTION <assertion-name> CHECK <predicate>
- E.g. The department id of manager relation is always not null since each manager works at least one department
- CREATE ASSERTION nomanager CHECK

  (NOT EXISTS (SELECT * FROM manager WHERE deptid = NULL))

It assures that there is no manager who is not assigned any departments at any time.

mid	mname	deptid
m1	Ram	d1
m2	Sita	d2
m3	Hari	NULL
Relation: manager		

Here, deptid of manager Hari is NULL, so assertion is violated and we cannot further modify database.

Chapter 5 : Database Constraints and Relational Database Design

- **Triggers:**
- is a stored procedure in database which automatically invokes whenever a special event in the database occurs. Like a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.
- Syntax:

CREATE TRIGGER [trigger-name] => *creates a trigger with trigger name*

[before | after] => *specifies when trigger will be executed*

[insert | update | delete] => *specifies DML operation*

ON table-name => *specifies name of table associated with trigger*

[FOR EACH ROW] => *specifies row level trigger*

[trigger-body] => *provides the operation to be performed as trigger is fired*

Chapter 5 : Database Constraints and Relational Database Design

- **Triggers:**
- Before triggers run the trigger action before the triggering statement is run.
- After trigger run the trigger action after the triggering statement is run.
- E.g. student (id,name,marks1,marks2,marks3,total,average)

CREATE TRIGGER stud-marks

BEFORE INSERT

ON student

FOR EACH ROW

SET new.total = new.marks1 + new.marks2 + new.marks3 , new.average = new.total/3

Chapter 5 : Database Constraints and Relational Database Design

- **Triggers:**
- INSERT INTO student VALUES (1,"Ram",40,50,90,0,0)
- SELECT * FROM student

1	Ram	40	50	90	180	60

- **SQL Standard Isolation Levels:**
  - SQL isolation levels play a crucial role in ensuring data consistency and integrity when multiple transactions are executed concurrently.
  - **Read Phenomena:**
    - The standard *SQL 92* defines three read phenomena describing various issues that may happen when two transactions are executed concurrently with no transaction isolation in place.

eid	name	salary
1	Ram	10000
2	Sita	15000

Fig: employee table

Chapter 8: Transactions and Concurrency Control

- **SQL Standard Isolation Levels:**
  - **Read Phenomena:**
    - **Dirty Read**
      - When two transactions access the same data and we allow for reading values that are not yet committed, we may get a dirty read.

Transaction1	Transaction2
	UPDATE employee SET salary=15000 WHERE eid=1
SELECT salary FROM employee WHERE eid=1	
	ROLLBACK

- *Here,Transaction* 2 modifies row with *id = 1*, then *Transaction 1* reads the row and gets value *15000*, and *Transaction* 2 rolls things back. Effectively, *Transaction 1* uses value that doesn't exist in the database.

- **SQL Standard Isolation Levels:**
  - **Read Phenomena:**
    - **Non Repeatable Read**
      - Non Repeatable read is a problem when a transaction reads the same thing twice and gets different results each time.

Transaction1	Transaction2
SELECT salary FROM employee WHERE eid=1	
	UPDATE employee SET salary=15000 WHERE eid=1 COMMIT
SELECT salary FROM employee WHERE eid=1	

- *Transaction 1* reads a row and gets value *10000*. *Transaction 2* modifies the same row. Then *Transaction 1* reads the row again and gets a different value (*15000* this time).

Chapter 8: Transactions and Concurrency Control

- **SQL Standard Isolation Levels:**
  - **Read Phenomena:**
    - **Phantom Read**
      - Phantom read is a case when a transaction looks for rows the same way twice but gets different results.

Transaction1	Transaction2
SELECT * FROM employee WHERE salary < 20000	
	INSERT INTO employee VALUES (3,"Hari",10000) COMMIT
SELECT * FROM employee WHERE salary<20000	

- *Transaction* 1 reads rows and finds two of them matching the conditions. *Transaction* 2 adds another row that matches the conditions used by the *Transaction* 1. When the *Transaction* 1 reads again, it gets a different set of rows. We would expect to get the same rows for both *SELECT* statements of *Transaction* 1.

Chapter 8: Transactions and Concurrency Control

- **SQL Standard Isolation Levels:**
  - **ISOLATION LEVELS:**
    - There are 4 standard levels: *READ UNCOMMITED*, *READ COMMITED*, *REPEATABLE READ*, and *SERIALIZABLE*.
    - ***READ UNCOMMITED*** allows a transaction to read data that is not yet committed to the database. This allows for highest performance but it also leads to most undesired read phenomena.
    - ***READ COMMITED*** allows a transaction to read only data that is committed. This avoids issue of reading data that "later disappears" but doesn't protect from other read phenomena.
    - ***REPEATABLE READ*** level tries to avoid issue of reading data twice and getting different results.
    - Finally, ***SERIALIZABLE*** tries to avoid all read phenomena.

Chapter 8: Transactions and Concurrency Control

- **SQL Standard Isolation Levels:**
  - **ISOLATION LEVELS:**

Level \ Phenomena	Dirty Read	Repeatable Read	Phantom Read
READ UNCOMMITED	YES	YES	YES
READ COMMITED	NO	YES	YES
REPEATABLE READ	NO	NO	YES
SERIALIZABLE	NO	NO	NO

Chapter 8 : Transactions and Concurrency Control

- **Graph based Protocol:**
  - These protocols offer an alternative to traditional locking mechanisms, often providing better performance and scalability, especially in high-contention environments.
- **Advantages of Graph-Based Protocols**
  - **Improved Concurrency:** By allowing more flexibility in the order of transaction execution, graph-based protocols can often achieve higher concurrency levels than traditional locking protocols.
  - **Reduced Lock Contention:** Since transactions do not necessarily need to acquire locks on all data items they access, lock contention can be reduced.
  - **Scalability:** Graph-based protocols can scale well to large numbers of concurrent transactions.

Chapter 8 : Transactions and Concurrency Control

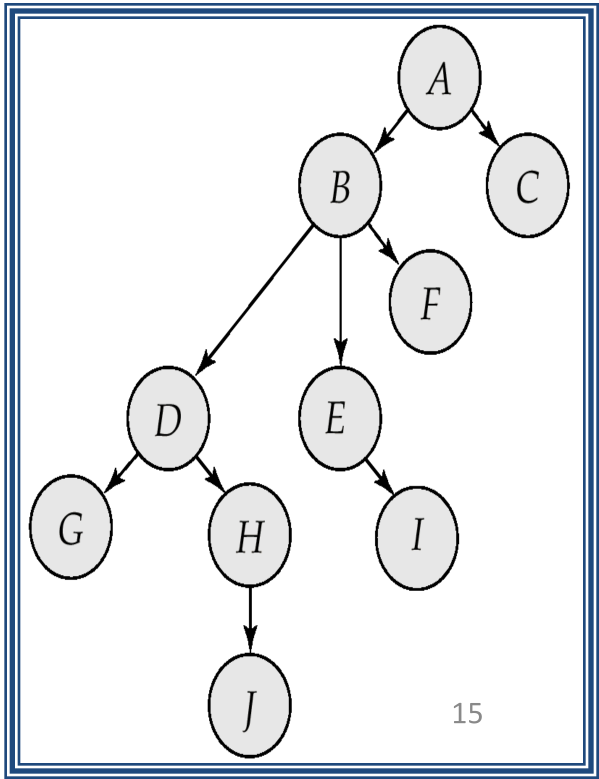
- **How Graph-Based Protocols Work**
  - **Transaction Submission:** When a transaction is submitted, it is assigned a timestamp.
  - **Conflict Detection:** As transactions execute, conflicts are detected and recorded in the transaction graph.
  - **Cycle Detection:** The graph is periodically checked for cycles. If a cycle is found, it indicates a deadlock, and one or more transactions must be aborted to break the cycle.
  - **Serializability Enforcement:** The protocol ensures that the final schedule is equivalent to a serial schedule by enforcing a partial order on the transactions based on the timestamp and dependency information in the graph.

Graph Based Protocol

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, ..., d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph (DAG), called a *database graph*.

- The *tree-protocol* is a simple kind of graph protocol.
  - Only exclusive locks are allowed.
  - The first lock by T_i may be on any data item if there is no lock on the data item.
  - Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
  - Data items may be unlocked at any time.

- in tree locking protocol a transaction may have to lock data items that it does not access.
- increased locking overhead and additional waiting time
- potential decrease in concurrency
- it is deadlock free so no rollback
- unlocking may occur earlier



	T1	T2	T3
1	**Lock-X(A)**		
2	**Lock-X(B)**		
3		**Lock-X(D)**	
4		**Lock-X(H)**	
5		**UnLock-X(D)**	
6	**Lock-X(E)**		
7	**Lock-X(D)**		
8	**UnLock-X(B)**		
9	**UnLock-X(E)**		
10			**Lock-X(B)**
11			**Lock-X(E)**
12		**UnLock-X(H)**	
13	**Lock-X(B)**		
14	**Lock-X(G)**		
15	**UnLock-X(D)**		
16			**UnLock-X(E)**
17			**UnLock-X(B)**
18	**UnLock-X(G)**		

	T1	T2	T3
1	**Lock-X(A)**		
2	**Lock-X(B)**		
3		**Lock-X(D)**	
4		**Lock-X(H)**	
5		**UnLock-X(D)**	
6	**Lock-X(E)**		
7	**Lock-X(D)**		
8	**UnLock-X(B)**		
9	**UnLock-X(E)**		
10			**Lock-X(B)**
11			**Lock-X(E)**
12		**UnLock-X(H)**	
13	**Lock-X(G)**		
14	**UnLock-X(D)**		
15			**UnLock-X(E)**
16			**UnLock-X(B)**
17	**UnLock-X(G)**		