Module 3: Transaction Processing

- **Transaction = Unit of Work**
  - Recall ACID Properties (from Module 1)
- **Requirements of Transactions in a DBMS**
  - 7-by-24 access
  - Concurrency Control
  - Recovery
- **Example: Transfer $50 from Account A to Account B**
  - Pseudocode:

```
Transaction 1:
        A.balance = Read(A);
        A.balance = A.balance - 50;
        Write(A, A.balance)
        B.balance = Read(B);
        B.balance = B.balance + 50;
        Write(B, B.balance);
```

- **Result of a Transaction**
  - Commit (Success)
  - Rollback (Abort)

Concurrency Control

- **When is it applicable?**
  - Not for A,C or D
- **Comes into effect when Isolation is considered**
  - Example:

```
Transaction 2:
        A.balance = Read(A);
        B.balance = Read(B);
        Print(A.balance+B.balance);
```

  - What happens if Transaction 2 starts before 1 finishes?
- **How do you enforce “isolation”?**
  - Serializability: The property that a group of transactions has the same effect and output as some serial execution of the transactions

Example

- **Consider Transactions 1 and 2**
  - Form a “schedule” of operations
- **Schedule for Transaction 1**
  - $r_1[A]$, $w_1[A]$, $r_1[B]$, $w_1[B]$
- **Schedule for Transaction 2**
  - $r_2[A]$, $r_2[B]$
- **Correct Way to Interleave them**
  - $r_1[A]$, $w_1[A]$, $r_2[A]$, $r_1[B]$, $w_1[B]$, $r_2[B]$
- **Also correct (but really it is only serial)**
  - $r_1[A]$, $w_1[A]$, $r_1[B]$, $w_1[B]$, $r_2[A]$, $r_2[B]$
- **Wrong Way to Interleave them**
  - $r_1[A]$, $w_1[A]$, $r_2[A]$, $r_2[B]$, $r_1[B]$, $w_1[B]$
  - Why?: because $50 is lost from the report

Back to Basics

- **What is a Schedule?**
  - Time-Ordered Sequence of Important Actions taken by 1 or more Txs
- **What is a Serial Schedule?**
  - One which consists of all actions of one Tx, followed by all actions of another, and so on.
- **What is a Serializable Schedule?**
  - One whose effect on the DB “state” is the same as that of some serial schedule
  - Notice the word “some”
  - All serial schedules are serializable, but not necessarily the reverse!
- **In general, serializability is a good property, but**
  - difficult to achieve due to lack of effective algorithms
- **Compromise: Conflict-Serializability**
  - A restricted notion of serializability
  - A sufficient but not necessary condition for serializability

What is Conflict-Serializability?

- **Basic Idea: Start with a given schedule**
  - Swap neighboring entries, if they don't conflict
  - See if you obtain a serial schedule, continuing in this manner
- **When can Neighboring Entries be Swapped?**
  - Almost always, except when they involve the same DB element and one of them is a write
- **Example**
  - $r_1(A)\ w_1(A)\ r_2(A)\ \underline{w_2(A)\ r_1(B)}\ w_1(B)\ r_2(B)\ w_2(B)$
  - $r_1(A)\ w_1(A)\ \underline{r_2(A)\ r_1(B)}\ w_2(A)\ w_1(B)\ r_2(B)\ w_2(B)$
  - $r_1(A)\ w_1(A)\ r_1(B)\ r_2(A)\ \underline{w_2(A)\ w_1(B)}\ r_2(B)\ w_2(B)$
  - $r_1(A)\ w_1(A)\ r_1(B)\ \underline{r_2(A)\ w_1(B)}\ w_2(A)\ r_2(B)\ w_2(B)$
  - $r_1(A)\ w_1(A)\ r_1(A)\ w_1(B)\ r_2(A)\ w_2(A)\ r_2(B)\ w_2(B)$
- **Can Conflict-Serializability determine all Serializable Schedules?**
  - Answer: No! (see previous slide)
  - Example, the two schedules are equivalent according to serializability but not according to conflict-serializability
    - $w_1(B)\ w_1(A)\ w_2(B)\ w_2(A)\ w_3(A)$
    - $w_1(B)\ w_2(B)\ w_2(A)\ w_1(A)\ w_3(A)$

A Calculation Example

- **Consider**
  - T1: READ(A,t); t=t+2; WRITE(A,t); READ(B,t); t=t*3; WRITE(B,t)
  - T2: READ(B,s); s=s*2; WRITE(B,s); READ(A,s); s=s+3; WRITE(A,s)
- **Considering Reads and Writes Only**
  - T1: $r_1(A)\ w_1(A)\ r_1(B)\ w_1(B)$
  - T2: $r_2(B)\ w_2(B)\ r_2(A)\ w_2(A)$
- **Calculations**
  - Schedules (all possible interleavings) = ${}^8C_4$
  - Serial Schedules: 2
  - Serializable Schedules: $2 + {}^4C_2 * {}^4C_2$
  - Conflict-Serializable Schedules: 2
- **Notice that..**
  - we have taken advantage of the information from the first bullet
  - even though it isn't represented in the second bullet
- **Assumption**
  - Operations on "A" are commutative (+)
  - Operations on "B" are commutative (*)

What we know so far

- **Conflict-Serializability is not a test for Serializability**
- **Is there a test for Conflict-Serializability?**
  - Answer: Yes
- **A Graph-Theoretic Construction**
  - Add a node for every Tx
  - Add a directed edge from Tx to Ty if there is some operation in Tx that takes precedence over some action in Ty
  - Ask: Is the graph cyclic?
  - Yes?: Schedule is not Conflict-Serializable
- **In other words, a topological sort**
  - Can be achieved by doing a DFS
  - Or removing nodes (and edges) that have least indegree repeatedly
- **How is Conflict-Serializability Enforced?**
  - Answer: Locks

Locking

- **Assumptions**
  - Consistency of Transactions: Tx should get a lock before reading or writing an element; should release lock when done
  - Legality of Schedules: No two Txs can have locked the same element
- **Notation**
  - $l_1(A)$: for locking
  - $u_1(A)$: for unlocking
- **Two-approaches to insert "l" and "u": Consider**
  - T1: $r_1(A)$ $w_1(A)$ $r_1(B)$ $w_1(B)$
  - T2: $r_2(A)$ $w_2(A)$ $r_2(B)$ $w_2(B)$
- **A "Nice" Approach": Unlock as soon as possible**
  - T1: $l_1(A)$ $r_1(A)$ $w_1(A)$ $u_1(A)$ $l_1(B)$ $r_1(B)$ $w_1(B)$ $u_1(B)$
  - T2: $l_2(A)$ $r_2(A)$ $w_2(A)$ $u_2(A)$ $l_2(B)$ $r_2(B)$ $w_2(B)$ $u_2(B)$
- **A "Greedy" Approach": Acquire all locks first, then unlock**
  - T1: $l_1(A)$ $r_1(A)$ $w_1(A)$ $l_1(B)$ $u_1(A)$ $r_1(B)$ $w_1(B)$ $u_1(B)$
  - T2: $l_2(A)$ $r_2(A)$ $w_2(A)$ $l_2(B)$ $u_2(A)$ $r_2(B)$ $w_2(B)$ $u_2(B)$
  - Problem: T2 has to wait for T1

Which Approach is Preferable?

- **Surprise: The Latter!**
  - Called Two-Phase Locking (2PL)
  - All lock requests precede all unlock requests
  - First Phase: Growing Phase (Acquire Locks)
  - Second Phase: Shrinking Phase (Release Locks)
- **Why is this preferable?**
  - Ensures conflict-serializability
- **What is the conflict-equivalent serial schedule?**
  - same as the ordering of the transactions acc. to their first unlocks
  - why?: Each 2PL Tx can be assumed to execute in its entirety at the instant it issues its first unlock
- **Other issues in 2PL**
  - Foolproof?: No, possibility for Deadlock
  - Types of Locks
  - Granularity of Locks

Types of Locks

- **Shared Locks (Read access) and Exclusive Locks (Write Access)**
  - $sl_1(A)$: for shared locks
  - $xl_1(A)$: for exclusive locks
  - $u_1(A)$: for unlocking (same as previous)
- **Compatibility Matrix**

Table 1: Matching Locks

	S	X
S	yes	no
X	no	no

- **Can the same Tx hold both S and X locks on one element?**
  - Yes!: But X takes precedence
  - Notice that the compatibility matrix above is only for different Transactions!
- **Problems with S and X locks**
  - Can cause deadlock

Solution:

- **Upgrade/Update Locks (U)**
  - Looks like a shared lock when requested
  - can be upgraded to an X lock later
  - Effectively introduces asymmetry into the compatibility matrix

	S	X	U
S	yes	no	yes
X	no	no	no
U	no	no	no

- **Increment Locks (U)**
  - Commute with other increment locks

	S	X	I
S	yes	no	no
X	no	no	no
I	no	no	yes

Who Inserts Locks and Unlocks?

- **Not the requesting transactions themselves!**
  - Job of Transaction scheduler
- **Three Aspects of a Tx Scheduler**
  - Scheduler 1: Insert “l” and “u” according to locking schedule (e.g. 2PL)
  - Scheduler 2: See if any Txs have to wait for anybody, deny requests etc.
  - Scheduler 3: Check for strange situations, deadlock, dirty reads etc.
- **What is a Lock anyway?**
  - Modeled by a lock-table
  - Hash-table for mapping database elements to locking and waiting info.
- **Dominating Information from a Compatibility Matrix**
  - In S,X table: X dominates S
  - In S,X,U table: U dominates S, X dominates both S and U
  - useful for processing transaction requests

Locks with Multiple Granularities

- **Example: Can Lock**
  - Whole relation
  - Individual pages
  - Tuples etc.

- **Warning Locks: A Protocol for Hierarchical Structures**
  - Basic Idea: To read (S) deep down below, insert IS locks all the way from root to the element of interest
  - To write (X) deep down below, insert IX locks all the way from root to the element of interest
  - IS: “I intend to read something down below”
  - IX: “I intend to write something down below”

- **Compatibility Matrix**

	IS	IX	S	X
IS	yes	yes	yes	no
IX	yes	yes	no	no
S	yes	no	yes	no
X	no	no	no	no

Problems with previous slide

- **S and IX don’t dominate each other! (why?)**
  - Solution: Introduce a new locking mode (SIX)
  - Provide the conjunction of the two modes
  - Can be viewed as a mode in its own right: why? (a popular operation)

	IS	IX	S	X	SIX
IS	yes	yes	yes	no	yes
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
X	no	no	no	no	no
SIX	yes	no	no	no	no

- **To obtain**
  - IS or S -> must have on all ancestors: IS or IX
  - IX or X or SIX -> must have on all ancestors: IX or SIX

- **More Problems**
  - The Phantom Menace: Insertion of New Tuples
  - Should be viewed as an X operation

Recovery

- **When is it applicable?**
  - For A and D (of ACID)
- **Transactions are performed in a "tentative" manner**
  - For ensuring A, UNDO operation
  - For ensuring D, REDO operation
  - provided by the buffer manager
- **UNDO and Policies for implementing it**
  - Process of removing effects of an incomplete/aborted Tx for preserving A
  - STEAL: allowing updates made by an uncommitted Tx to overwrite the most recently committed value of a data item on nonvolatile storage
  - NO-STEAL: opposite of STEAL
- **REDO and Policies for implementing it**
  - Process of reinstating effects of an uncommitted Tx for preserving D
  - FORCE: ensuring that all updates made by a Tx are reflected on nonvolatile storage before the Tx is allowed to commit
  - NO-FORCE: opposite of FORCE
- **Which places fewest demands on UNDO and REDO recovery?**
  - NO-STEAL and FORCE

In Real Life ...

- **STEAL/NO-FORCE is used! Why?**
  - NO-STEAL obligates the retention of temporary data in swap areas
  - FORCE causes disk write overhead in the path of a committing transaction
  - WAL: Write-Ahead Logging - a policy for enforcing STEAL/NO-FORCE
  - Used by most commercial systems
- **How is "C" achieved (in ACID)**
  - Active and Rule-Based Elements (Constraints, Triggers)
  - Help encode application-specific constraints
  - allow DBMS to be reactive
- **Limitations of ACID Model**
  - Very restrictive; cannot model many important applications
  - e.g. mobile computing, collaborative computing, workflow-process mgmt.
  - too general for richer transaction semantics