Announcements

- HW2 deadline extended by 2.5 days -- to Saturday, 10/21, 5 pm
- Project midterm report deadline extended by 7 days -- to Wednesday, 11/01, 11:55 pm
- Keep working on your proposed project too
- To see your midterm, please come to my office between 11:50 am and 12:30 pm today

Where are we now?

- [RG] Chapter 16.1-16.3, 16.4.1
- 17.1-17.4
- 17.5.1, 17.5.3

Next

- Transactions
  - Basic concepts
  - Concurrency control
  - Recovery
  - (for the next 4-5 lectures)

Reading Material

Motivation: Concurrent Execution

- Concurrent execution of user programs is essential for good DBMS performance.
  - Disk accesses are frequent, and relatively slow
  - It is important to keep the CPU busy by working on several user programs concurrently
  - Short transactions may finish early if interleaved with long ones
  - May increase system throughput (avg. #transactions per unit time) and decrease response time (avg. time to complete a transaction)

- A user’s program may carry out many operations on the data retrieved from the database
  - But the DBMS is only concerned about what data is read/written from/to the database

Transactions

- A transaction is the DBMS’s abstract view of a user program
  - A sequence of reads and write statements
  - The same program executed multiple times would be considered as different transactions
  - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements
  - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed)
Example

• Consider two transactions:

\begin{align*}
T1: & \text{ BEGIN } A=A+100, \ B=B-100 \ \text{END} \\
T2: & \text{ BEGIN } A=1.06^*A, \ B=1.06^*B \ \text{END}
\end{align*}

• Intuitively, the first transaction is transferring $100 from B’s account to A’s account. The second is crediting both accounts with a 6% interest payment

• There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

• However, the net effect must be equivalent to these two transactions running serially in some order

Commit and Abort

\begin{align*}
T1: & \text{ BEGIN } A=A+100, \ B=B-100 \ \text{END} \\
T2: & \text{ BEGIN } A=1.06^*A, \ B=1.06^*B \ \text{END}
\end{align*}

• A transaction might commit after completing all its actions

• or it could abort (or be aborted by the DBMS) after executing some actions

Concurrency Control and Recovery

\begin{align*}
T1: & \text{ BEGIN } A=A+100, \ B=B-100 \ \text{END} \\
T2: & \text{ BEGIN } A=1.06^*A, \ B=1.06^*B \ \text{END}
\end{align*}

• Concurrency Control

– (Multiple) users submit (multiple) transactions

– Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions

– user should think of each transaction as executing by itself one-at-a-time

– The DBMS needs to handle concurrent executions

• Recovery

– Due to crashes, there can be partial transactions

– DBMS needs to ensure that they are not visible to other transactions

ACID Properties

• Atomicity

• Consistency

• Isolation

• Durability

Atomicsity

\begin{align*}
T1: & \text{ BEGIN } A=A+100, \ B=B-100 \ \text{END} \\
T2: & \text{ BEGIN } A=1.06^*A, \ B=1.06^*B \ \text{END}
\end{align*}

• A user can think of a transaction as always executing all its actions in one step, or not executing any actions at all

– Users do not have to worry about the effect of incomplete transactions
Consistency

- Each transaction, when run by itself with no concurrent execution of other actions, must preserve the consistency of the database
  - e.g. if you transfer money from the savings account to the checking account, the total amount still remains the same

```plaintext
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
```

Isolation

- A user should be able to understand a transaction without considering the effect of any other concurrently running transaction
  - even if the DBMS interleaves their actions
  - transaction are “isolated or protected” from other transactions

```plaintext
T1: BEGIN A=A+100, B=B-100 END
T2:BEGIN A=1.06*A, B=1.06*B END
```

Durability

- Once the DBMS informs the user that a transaction has been successfully completed, its effect should persist
  - even if the system crashes before all its changes are reflected on disk

```plaintext
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
```

Ensuring Consistency

- e.g. Money debit and credit between accounts
- User’s responsibility to maintain the integrity constraints
- DBMS may not be able to catch such errors in user program’s logic
  - e.g. if the credit is (debit – 1)
- However, the DBMS may be in inconsistent state “during a transaction” between actions
  - which is ok, but it should leave the database at a consistent state when it commits or aborts
- Database consistency follows from transaction consistency, isolation, and atomicity

Ensuring Isolation

- DBMS guarantees isolation (later, how)
- If T1 and T2 are executed concurrently, either the effect would be T1→T2 or T2→T1 (and from a consistent state to a consistent state)
- But DBMS provides no guarantee on which of these order is chosen
- Often ensured by “locks” but there are other methods too

Ensuring Atomicity

- Transactions can be incomplete due to several reasons
  - Aborted (terminated) by the DBMS because of some anomalies during execution
    - in that case automatically restarted and executed anew
  - The system may crash (say no power supply)
  - A transaction may decide to abort itself encountering an unexpected situation
    - e.g. read an unexpected data value or unable to access disks
Ensuring Atomicity

- A transaction interrupted in the middle can leave the database in an inconsistent state
- DBMS has to remove the effects of partial transactions from the database
- DBMS ensures atomicity by “undoing” the actions of incomplete transactions
- DBMS maintains a “log” of all changes to do so

Ensuring Durability

- The log also ensures durability
- If the system crashes before the changes made by a completed transaction are written to the disk, the log is used to remember and restore these changes when the system restarts
- “recovery manager” will be discussed later
  - takes care of atomicity and durability

Notations

- Transaction is a list of “actions” to the DBMS
  - includes “reads” and “writes”
  - \( \text{R}_T(O) \): Reading an object \( O \) by transaction \( T \)
  - \( \text{W}_T(O) \): Writing an object \( O \) by transaction \( T \)
  - also should specify \( \text{Commit}_T(C_T) \) and \( \text{Abort}_T(A_T) \)
  - \( T \) is omitted if the transaction is clear from the context

Assumptions

- Transactions communicate only through READ and WRITE
  - i.e. no exchange of message among them
- A database is a fixed collection of independent objects
  - i.e. objects are not added to or deleted from the database
  - this assumption can be relaxed
    - (dynamic db/phantom problem later)

Schedule

- An actual or potential sequence for executing actions as seen by the DBMS
- A list of actions from a set of transactions
  - includes READ, WRITE, ABORT, COMMIT
- Two actions from the same transaction \( T \) MUST appear in the schedule in the same order that they appear in \( T \)
  - cannot reorder actions from a given transaction

Serial Schedule

- If the actions of different transactions are not interleaved
  - transactions are executed from start to finish one by one
Problems with a serial schedule

- The same motivation for concurrent executions, e.g.,
  - while one transaction is waiting for page I/O from disk, another transaction could use the CPU
  - reduces the time disks and processors are idle
- Increases system throughput
  - average transactions completed in a given time
- Also improves response time
  - average time taken to complete a transaction
  - since short transactions can be completed with long ones and do not have to wait for them to finish

Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions
- Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule
- Serializable schedule: A schedule that is equivalent to some serial execution of the committed transactions
  - Note: If each transaction preserves consistency, every serializable schedule preserves consistency

Serializable Schedule

- If the effect on any consistent database instance is guaranteed to be identical to that of "some" complete serial schedule for a set of "committed transactions"
- However, no guarantee on T1 > T2 or T2 > T1

Anomalies with Interleaved Execution

- If two consistency-preserving transactions when run interleaved on a consistent database might leave it in inconsistent state
  - Write-Read (WR)
  - Read-Write (RW)
  - Write-Write (WW)
- No conflict with RR if no write is involved

WR Conflict

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):
  - transaction T2 reads an object that has been modified by T1 but not yet committed
  - or T2 reads an object from an inconsistent database state (like fund is being transferred between two accounts by T1 while T2 adds interest to both)

RW Conflict

- Unrepeatable Reads (RW Conflicts):
  - T2 changes the value of an object A that has been read by transaction T1, which is still in progress
  - If T1 tries to read A again, it will get a different result
  - Suppose two customers are trying to buy the last copy of a book simultaneously
**WW conflict**

- Overwriting Uncommitted Data (WW Conflicts, "lost update"):
  - T2 overwrites the value of A, which has been modified by T1, still in progress
  - Suppose we need the salaries of two employees (A and B) to be the same
    - T1 sets them to $1000
    - T2 sets them to $2000

**Schedules with Aborts**

- Actions of aborted transactions have to be undone completely
  - may be impossible in some situations
    - say T2 reads the fund from an account and adds interest
  - T1 aims to deposit money but aborts
  - if T2 has not committed, we can "cascade aborts" by aborting T2 as well
  - if T2 has committed, we have an "unrecoverable schedule"

**Recoverable Schedule**

- Transaction commits if and only after all transactions they read have committed
  - avoids cascading aborts

**Conflict Equivalent Schedules**

- Two schedules are conflict equivalent if:
  - involve the same actions of the same transactions
  - Every pair of conflicting actions of two committed transactions is ordered the same way

- Conflicting actions:
  - both by the same transaction Ti
  - both on the same object by two transactions Ti and Tj, at least one action is a write
    - R(X), W(X)
    - W(X), R(X)
    - W(X), W(X)

**Conflict Serializable Schedules**

- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

- In class:
  - \( r_1(A); w_1(A); r_2(A); w_2(A); r_3(B); w_3(B) \)
  - to
  - \( r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \)
Example

• A schedule that is not conflict serializable:

\[
\begin{align*}
T_1: & R(A), W(A), R(B), W(B) \\
T_2: & R(A), W(A), R(B), W(B)
\end{align*}
\]

\[R(A), W(A), R(A), W(A), R(B), W(B), R(B), W(B)\]

- The cycle in the graph reveals the problem.
- The output of T1 depends on T2, and vice-versa.

Precedence Graph

• Also called dependency graph, conflict graph, or serializability graph
• One node per committed transaction
• Edge from \(T_i\) to \(T_j\) if an action of \(T_i\) precedes and conflicts with one of \(T_j\)'s actions
- \(W(A)\) --- \(R(A)\), or
- \(R(A)\) --- \(W(A)\), or
- \(W(A)\) --- \(W(A)\)
• \(T_i\) must precede \(T_j\) in any serial schedule

Conflict Serializability

• Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic

\[R(A), W(A), R(A), W(A), R(B), W(B), R(B), W(B)\]

Lock-Based Concurrency Control

• DBMS should ensure that only serializable and recoverable schedules are allowed
- No actions of committed transactions are lost while undoing aborted transactions
• Uses a locking protocol
• Lock: a bookkeeping object associated with each "object"
- different granularity
• Locking protocol:
- a set of rules to be followed by each transaction

Strict two-phase locking (Strict 2PL)

Two rules
1. Each transaction must obtain
   - a S (shared) lock on object before reading
   - and an X (exclusive) lock on object before writing
   - exclusive locks also allow reading an object, additional shared lock is not required
   - If a transaction holds an X lock on an object, no other transaction can get a lock (S or X) on that object
   - transaction is suspended until it acquires the required lock
2. All locks held by a transaction are released when the transaction completes

Example: Strict 2PL

\[
\begin{align*}
T_1: & R(A), W(A), R(A), W(A), R(B), W(B), R(B), W(B), R(B), W(B), R(B), W(B) \\
T_2: & R(B), W(B), Commit
\end{align*}
\]

• WR conflict (dirty read)
• Strict 2PL does not allow this
Example: Strict 2PL

• Strict 2PL allows interleaving

More on Strict 2PL

• Every transaction has
  – a growing phase of acquiring locks, and
  – a shrinking phase of releasing locks

• Strict 2PL allows only serializable schedules
  – precedence graphs will be acyclic (check yourself)
  – Additionally, allows recoverable schedules and simplifies transaction aborts
  – two transactions can acquire locks on different objects independently

2PL vs. strict 2PL

• 2PL:
  – first, acquire all locks, release none
  – second, release locks, cannot acquire any other lock

• Strict 2PL:
  – release write (X) lock, only after it has ended (committed or aborted)

• (Non-strict) 2PL also allows only serializable schedules like strict 2PL, but involves more complex abort processing

Strict 2PL and Conflict Serializability

• Strict 2PL allows only schedules whose precedence graph is acyclic
• Can never allow cycles as the X locks are being held by one transaction
• However, it is sufficient but not necessary for serializability
• Relaxed solution: View serializability

View Serializability

• Schedules S1 and S2 are view equivalent if:
  – If Tᵢ reads initial value of A in S₁, then Tᵢ also reads initial value of A in S₂
  – If Tᵢ reads value of A written by Tⱼ in S₁, then Tᵢ also reads value of A written by Tⱼ in S₂
  – For all data object A, if Tᵢ writes final value of A in S₁, then Tᵢ also writes final value of A in S₂

• S is view serializable, if it is view equivalent to some serial schedule

More on View Serializability

• Every conflict serializable schedule is view serializable (check it yourself)
• But the converse may not be true
• If VS but not CS, would contain a “blind write” (see below)
• Verifying and enforcing VS is more expensive than CS, so less popular than CS
Lock Management

• Lock and unlock requests are handled by the lock manager
• Lock table entry:
  – Number of transactions currently holding a lock
  – Type of lock held (shared or exclusive)
  – Pointer to queue of lock requests (if the shared or exclusive lock cannot be granted immediately)
• Locking and unlocking have to be atomic operations
• Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
• Transaction commits or aborts
  – all locks released

Deadlocks

• Deadlock: Cycle of transactions waiting for locks to be released by each other
  – database systems periodically check for deadlocks
• Two ways of dealing with deadlocks:
  – Deadlock detection
  – Deadlock prevention

Deadlock Detection

1. Create a waits-for graph: (example on next slide)
   – Nodes are transactions
   – There is an edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ to release a lock
   • Periodically check for cycles in the waits-for graph
   • Abort a transaction on a cycle and release its locks, proceed with the other transactions
     – several choices
     – one with the fewest locks
     – one has done the least work/farthest from completion
     – if being repeatedly restarted, should be favored at some point
2. Use timeout, if long delay, assume (pessimistically) a deadlock

Deadlock Prevention

• Assign priorities based on timestamps
• Assume $T_i$ wants a lock that $T_j$ holds. Two policies are possible:
  – Wait-Die: if $T_i$ has higher priority, $T_i$ waits for $T_j$; otherwise $T_j$ aborts
  – Wound-wait: if $T_i$ has higher priority, $T_i$ aborts; otherwise $T_j$ waits
• Convince yourself that no cycle is possible
• If a transaction re-starts, make sure it has its original timestamp
  – each transaction will be the oldest one and have the highest priority at some point
• A variant of strict 2PL, conservative 2PL, works too
  – acquire all locks it ever needs before a transaction starts
  – no deadlock but high overhead and poor performance, so not used in practice

Summary

• Transaction
  – $R_i(A)$, $W_i(A)$, ...
  – Commit $C_i$, abort $A_i$
  – Lock/unlock: $S_i(A)$, $X_i(A)$, $US_i(A)$, $UX_i(A)$
• ACID properties
  – what they mean, whose responsibility to maintain each of them
• Conflicts: RW, WR, WW
• 2PL/Strict 2PL
  – all lock acquires have to precede all lock releases
  – Strict 2PL: release X locks only after commit or abort
Summary

- Schedule
  - Serial schedule
  - Serializable schedule (why do we need them?)
  - Conflicting actions
  - Conflict-equivalent schedules
  - Conflict-serializable schedule
  - View serializable schedule (relaxation)
  - Conflict Serializability => View Serializability => Serializability
  - Recoverable schedules

- Dependency (or Precedence) graphs
  - their relation to conflict serializability (by acyclicity)
  - their relation to Strict 2PL

Summary

- Lock management basics
- Deadlocks
  - detection
    - waits-for graph has cycle, or timeout
    - what to do if deadlock is detected
  - prevention
    - wait-die and wound-wait