CompSci 516
Database Systems

Lecture 13
Intro to Transactions

Instructor: Sudeepa Roy
Where are we now?

We learnt
✓ Relational Model and Query Languages
  ✓ SQL, RA, RC
  ✓ Postgres (DBMS)
    ▪ HW1
✓ Database Normalization
✓ DBMS Internals
  ✓ Storage
  ✓ Indexing
    ○ Query Evaluation
    ○ Operator Algorithms
    ○ External sort
    ○ Query Optimization
✓ Map-reduce and spark
  ▪ HW2

Next

• Transactions
  – Basic concepts
  – Concurrency control
  – Recovery
  – (for the next 3-4 lectures)
Announcements (Thurs, 2/17)

• HW2 can be solved in a team of 2 from the start
  – Divide your project team into two groups
  – Let us know if you do not have someone to work with
  – Due on 3/1
  – Submission instructions to be posted
  – Start early – new framework will take time
  – Part-2 will be running your code on cloud

• Project updates from next week
  – See instructions on gradescope
Reading Material

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

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
Transactions

• 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

• A transaction is the DBMS’s abstract view of a user program
  – a sequence of reads and write
  – the same program executed multiple times would be considered as different transactions
  – Beyond enforcing some integrity constraints, 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) – it only cares about “read” and “write” sequences
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)

• 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
  – short transactions can be completed with long ones and do not have to wait for them to finish
Example

• Consider two transactions:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: BEGIN A=A+100, B=B-100 END</td>
<td></td>
</tr>
<tr>
<td>T2: BEGIN A=1.06<em>A, B=1.06</em>B END</td>
<td></td>
</tr>
</tbody>
</table>

• 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.
Example

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

**Consider a possible interleaving (schedule):**

| T1: | A=A+100, B=B-100 |
| T2: | A=1.06*A, B=1.06*B |

**This is OK. But what about:**

| T1: | A=A+100, B=B-100 |
| T2: | A=1.06*A, B=1.06*B |

**The DBMS’s view of the second schedule:**

| T1: | R(A), W(A), R(B), W(B) |
| T2: | R(A), W(A), R(B), W(B) |
Commit and Abort

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

• A transaction might commit after completing all its actions
• or it could abort (or be aborted by the DBMS) after executing some actions
ACID Properties

• Atomicity
• Consistency
• Isolation
• Durability
Atomicity

- 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

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

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

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

• 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

Next, how we maintain all these four properties
But, in detail later
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 transactions 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”
  - \( R_T(O) \): Reading an object \( O \) by transaction \( T \)
  - \( W_T(O) \): Writing an object \( O \) by transaction \( T \)
  - also should specify \( \text{Commit}_T \) (or \( C_T \)) and \( \text{Abort}_T \) (or \( A_T \))
  - \( T \) is omitted if the transaction is clear from the context

\[
T1: \text{BEGIN} \quad A = A + 100, \quad B = B - 100 \quad \text{END}
\]
\[
T2: \text{BEGIN} \quad A = 1.06 \times A, \quad B = 1.06 \times B \quad \text{END}
\]
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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

• If the actions of different transactions are not interleaved
  – transactions are executed from start to finish one by one

• Problems
  – May decrease throughput and increase response time
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

- Either equivalent to T1→T2 or to T2→T1

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td>COMMI</td>
</tr>
<tr>
<td>W(B)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

serial schedule

serializable schedules
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 interests to both)
• **Unrepeateable 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

<table>
<thead>
<tr>
<th>T1: R(A), R(A), W(A), C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A), C</td>
</tr>
</tbody>
</table>
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”

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A), Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A) Commit</td>
</tr>
</tbody>
</table>
# Recoverable Schedule

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

<table>
<thead>
<tr>
<th>T1: R(A), W(A),</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A), R(B), W(B), Commit</td>
<td></td>
</tr>
</tbody>
</table>

Example of Unrecoverable schedule
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 $T_i$
    • $R_i(X), W_i(Y)$
  – both on the same object by two transactions $T_i$ and $T_j$, at least one action is a write
    • $R_i(X), W_j(X)$
    • $W_i(X), R_j(X)$
    • $W_i(X), W_j(X)$
Conflict Equivalent Schedules

• Two conflict equivalent schedules have the same effect on a database
  – all pairs of conflicting actions are in same order
  – one schedule can be obtained from the other by swapping “non-conflicting” actions
    • either on two different objects
    • or both are read on the same object
Conflict Serializable Schedules

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

• In class:
  • \( r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \)
  • to
  • \( r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \)
Conflict Serializable Schedules

• Schedule S is *conflict serializable* if S is conflict equivalent to some serial schedule

• In class:
  • \( r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B) \)
  • to
  • \( r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B) \)
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
  - $W_i(A) \rightarrow R_j(A)$, or $R_i(A) \rightarrow W_j(A)$, or $W_i(A) \rightarrow W_j(A)$
- $T_i$ must precede $T_j$ in any serial schedule

- A schedule that is not conflict serializable:

  \[ R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B) \]

- The cycle in the graph reveals the problem. The output of $T_1$ depends on $T_2$, and vice-versa.
Conflict Serializability

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

\[
\begin{align*}
R_1(A), \ W_1(A), \ R_2(A), \ W_2(A), \ R_2(B), \ W_2(B), \ R_1(B), \ W_1(B)
\end{align*}
\]

\[
\begin{align*}
T1 & \rightarrow T2 \\
A & \rightarrow B
\end{align*}
\]

\[
\begin{align*}
r_1(A); \ w_1(A); \ r_2(A); \ w_2(A); \ r_1(B); \ w_1(B); \ r_2(B); \ w_2(B)
\end{align*}
\]
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

T1: \( R(A), W(A), \) \( R(B), W(B), \) Commit
T2: \( R(A), W(A), R(B), W(B), \) Commit

- WR conflict (dirty read)
- Strict 2PL does not allow this

T1: \( X(A), R(A), W(A), \)
T2: \( \) HAS TO WAIT FOR LOCK ON A

T1: \( X(A), R(A), W(A), X(B), R(B), W(B), C, UX(A), UX(B) \)
T2: \( X(A), R(A), W(A), X(B), R(B), W(B), C \)

All locks released here
Can use UX(A), UX(B) – for shared lock unlocking, US(A), US(B)
Example: Strict 2PL

T1: S(A), R(A), X(C), R(C), W(C), C
T2: S(A), R(A), X(B), R(B), W(B), C

• 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)
  – Also, allows recoverable schedules and simplifies transaction aborts
  – two transactions can acquire locks on different objects independently
  – But there may be “serializable” schedules that are NOT “conflict serializable”

\[
\begin{align*}
S1 \text{ (not conflict serializable)} & \equiv S2 \text{ (serial)} \\
T1: & \ R(A) \quad W(A) \ C \\
T2: & \quad W(A) \ C \\
T3: & \quad W(A) \ C \\
T1: & \ R(A) , W(A) \ C \\
T2: & \quad W(A) \ C \\
T3: & \quad W(A) \ C
\end{align*}
\]
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
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, e.g., with fewest locks that has done the least work
     - if being repeatedly restarted, should be favored at some point

2. Use timeout, if long delay, assume (pessimistically) a deadlock
Deadlock Detection

**Example:**

T1: S(A), R(A), S(B)
T2: X(B), W(B) X(C)
T3: S(C), R(C)
T4: X(B)
Deadlock Detection

Example:

T1:  S(A), R(A), S(B)
T2:  X(B), W(B), X(C)
T3:  S(C), R(C), X(A)
T4:  X(B)

Duke CS, Spring 2022
CompSci 516: Database Systems
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 $T_i$ aborts
  – **Wound-wait:** If $T_i$ has higher priority, $T_j$ aborts; otherwise $T_i$ 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
Summary

• Transaction
  – $R_1(A)$, $W_2(A)$, ...
  – Commit $C_1$, abort $A_1$
  – Lock/unlock: $S_1(A)$, $X_1(A)$, $US_1(A)$, $UX_1(A)$

• 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
  – Recoverable schedules
  – Cascade delete

• **Dependency (or Precedence) graphs**
  – their relation to conflict serializability (by acyclicitiy)
  – 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