Transactions

Introduction to Databases
CompSci 316 Fall 2022
Announcements (Tue. Nov 15)

• Gradiance-6—Query Processing, will be open until Friday Nov 18 5 pm (NOTE THE TIME).
  • See the post on Ed- nested loop join use (M-1) instead of (M-2) – book/gradiance does not consider output page
  • This is only for this gradiance, in this course and final exam, we will count output page as on the slides

• HW6-QP/JSON due Saturday Nov 19 10 pm
  • You need to know JSON/MongoDB only for this HW, not included in Final exam (Note: XML is included in Final)

• You are done with all homeworks with HW6 (Yay!) – you will have a practice problem set on Transactions – treat is very seriously as an ungraded homework, transactions is an important topic for the final exam.

• Use the remaining time to complete your project, there will be 1-2 more gradiances too.

• Project should be ready by the last week of classes – more info will be posted soon

• A really short 1-question feedback survey to be posted today if you want to share any concern/feedback with us where we can help – just write “None” otherwise (part of communication 2%).
Where are we now?

Relational model and queries
- Relational Model
- Query in SQL
- Query in RA

Database Design
- E/R diagram (design from scratch)
- Normal Forms (refine design)

Beyond Relational Model
- XML
- NOSQL JSON/MongoDB

DBMS Internals and Query Processing
- Storage
- Index
- Execution/Optimization

Transactions
- Basics
- Concurrency Control
- Recovery

Other topics
- Map-Reduce Parallel DBMS Distributed DBMS
- Recursion
- Data Mining
So far: **One query/update**

**One machine**

- Multiple query/updates
  - One machine

**Transactions**

- One query/update
  - Multiple machines

**Parallel query processing**

- Map-Reduce, Spark, ..

**Distributed query processing**

- We will see some of these briefly

- Multiple query/updates, multiple machines:
  - Distributed transactions, Two-Phase Commit protocol, ..
Why should we care about running multiple queries/updates/programs on a machine concurrently?
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)
- May decrease response time (avg. time to complete a transaction)
Transactions

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

- A **transaction** is the DBMS’s abstract view of a user program
- a sequence of reads and write
  - DBMS only cares about R/W of “elements” (tuples, tables, etc)
- the same program executed multiple times would be considered as different transactions
Example

• Consider two transactions:

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

• 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.
Are these interleaving (schedule) good?

<table>
<thead>
<tr>
<th>Schedule 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1:</strong> BEGIN A=A+100, B=B-100  END</td>
</tr>
<tr>
<td><strong>T2:</strong> BEGIN A=1.06<em>A, B=1.06</em>B END</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1:</strong> A=A+100, B=B-100</td>
</tr>
<tr>
<td><strong>T2:</strong> A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1:</strong> A=A+100, B=B-100</td>
</tr>
<tr>
<td><strong>T2:</strong> A=1.06<em>A, B=1.06</em>B</td>
</tr>
</tbody>
</table>
Example: View of DBMS

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

- **Schedule 2:**

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

- The DBMS’s view (and Notations!):

| T1:               | R(A), W(A), R(B), W(B) |
| T2:               | R(A), W(A), R(B), W(B) |

| R₁(A), W₁(A), R₂(A), W₂(A), R₂(B), W₂(B), R₁(B), W₁(B) |

- C₁ = “Commit” by Transaction T1.
- A₁ = “Abort” by Transaction T1

- Two possible representation of schedules
- No message passing
- Fixed set of objects (for now)
Commit and Abort

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

```
T1: BEGIN   A=A+100,   B=B-100   END
T2: BEGIN   A=1.06*A,   B=1.06*B   END
```
Concurrency Control and Recovery

• **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
  • Also, there can be some “completed” transactions with updated data still in memory (not yet to disk) and therefore lost in a crash
    • DBMS needs to ensure that the updates eventually go to disk

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

• Atomicity
• Consistency
• Isolation
• Durability

Recall our Disk-memory diagram!

Remember: 
COMMIT ≠ DISK WRITE

• Uncommitted transactions may write to disk (need to UNDO if they abort)

• Committed transactions may have changes still in memory unwritten to disk yet (will be “lost” in a crash, need to REDO)
**Atomicity**

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

- 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

Transactions can be aborted (terminated) by the DBMS or by itself:
- because of some anomalies during execution (and then restarts)
- the system may crash (say no power supply)
- may decide to abort itself encountering an unexpected situation (e.g. read an unexpected data value or unable to access disks)

Ensured by recovery methods using “Logs” by “undo”-ing 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

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

Responsibility of programmer’s code and ensured by DBMS through other properties
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

Often ensured by “Locks”, and other concurrency control approaches
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

Ensured by recovery methods using “Logs” by “redo”-ing complete/committed tr.

T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
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
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.
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></td>
</tr>
</tbody>
</table>

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

- Simple, but advantages of concurrent execution lost
Serializable Schedule

- Equivalent to “some” serial schedule
- However, no guarantee on T1-> T2 or T2 -> T1

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

serial schedule

serializable schedules

(Later, how to check for serializability)
Anomalies with Interleaved Execution

• Conflicts may arise if one transaction wants to write to a data that another transaction reads/writes

• **Write-Read (WR)** – reading uncommitted or “dirty” data
• **Read-Write (RW)** – unrepeatable reads
• **Write-Write (WW)** – overwriting uncommitted data or “lost updates”

• No conflict with RR if no write is involved
SQL transactions

- A transaction is automatically started when a user executes an SQL statement.
- Subsequent statements in the same session are executed as part of this transaction:
  - Statements see changes made by earlier ones in the same transaction.
  - Statements in other concurrently running transactions do not.
- `COMMIT` command commits the transaction:
  - Its effects are made final and visible to subsequent transactions.
- `ROLLBACK` command aborts the transaction:
  - Its effects are undone.
Fine prints

• Schema operations (e.g., CREATE TABLE) implicitly commit the current transaction

• Many DBMS support an AUTOCOMMIT feature, which automatically commits every single statement
  • You can turn it on/off through the API
SQL isolation levels

• Strongest isolation level: **SERIALIZABLE**
  • Complete isolation
• Weaker isolation levels:
  • **REPEATABLE READ,**
  • **READ COMMITTED,**
  • **READ UNCOMMITTED**
• Increase performance by eliminating overhead and allowing higher degrees of concurrency
• Trade-off: sometimes you get the “wrong” answer
“READ UNCOMMITTED”

• Can read “dirty” data (WR conflict)
  • A data item is dirty if it is written by an uncommitted transaction

• Problem: What if the transaction that wrote the dirty data eventually aborts?

• Example: wrong average

  -- T1:
  UPDATE User
  SET pop = 0.99
  WHERE uid = 142;

  SELECT AVG(pop)
  FROM User;

  ROLLBACK;

  T1: R(A), W(A), R(B), W(B), Abort

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

  -- T2:
  SELECT AVG(pop)
  FROM User;

  COMMIT;

Other examples

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

• No dirty reads, but non-repeateable reads possible (RW conflicts)
  • Reading the same data item twice can produce different results

• Example: different averages
  • -- T1:
    ```sql
    UPDATE User
    SET pop = 0.99
    WHERE uid = 142;
    COMMIT;
    ```
  • -- T2:
    ```sql
    SELECT AVG(pop)
    FROM User;
    COMMIT;
    ```

Suppose two customers are trying to buy the last copy of a book simultaneously
Example: 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
“REPEATABLE READ”

• Reads are repeatable, but may see **phantoms**
  • When new values are “inserted”, safe with “updates”
• Example: different average (still!)

• -- T1:
  INSERT INTO User
  VALUES(789, 'Nelson',
         10, 0.1);
  COMMIT;

• -- T2:
  SELECT AVG(pop)
  FROM User;

  SELECT AVG(pop)
  FROM User;
  COMMIT;
Summary of SQL isolation levels

<table>
<thead>
<tr>
<th>Isolation level/anomaly</th>
<th>Dirty reads</th>
<th>Non-repeatable reads</th>
<th>Phantoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Impossible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Possible</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Impossible</td>
</tr>
</tbody>
</table>

• Syntax: At the beginning of a transaction, `SET TRANSACTION ISOLATION LEVEL isolation_level [READ ONLY | READ WRITE];`
  • READ UNCOMMITTED can only be READ ONLY

• PostgreSQL defaults to READ COMMITTED

End of lecture on 11/15 Tues
(some slides have Been updated and Revisited on 11/17)
Bottom line

• Group reads and dependent writes into a transaction in your applications
  • E.g., enrolling a class, booking a ticket

• Anything less than SERIALABLE is potentially very dangerous
  • Use only when performance is critical
  • READ ONLY makes weaker isolation levels a bit safer
Announcements (Thu. Nov 17)

• Gradiance-6—Query Processing, will be open until Friday Nov 18, 5 pm (NOTE THE TIME).
   • See the post on Ed- nested loop join use (M-1) instead of (M-2) – book/gradiance does not consider output page
   • This is only for this gradiance, in this course, hw, and final exam, we will count output page as on the slides

• HW6-QP/JSON – gradescope will be open until Nov 21 (Mon) 10 pm “without penalty” to everyone
   • Note: not a deadline extension, so no further STINF extensions
   • Try to finish by Sunday – OHs canceled next week (there might be one on Monday), Ed might be slow on Monday
   • Ed will closed during the thanksgiving break
   • You need to know JSON/MongoDB only for this HW, not included in Final exam (Note: XML is included in Final)

• 1-question final feedback survey due by tonight (Thursday 11/17 midnight). Just write “None” if no comments (part of communication 2%).
   • You can drag the textbox from the bottom-right corner to make it bigger!

• Check out the post on Ed for project presentations and final report (important)
   • Keep working on your project – no more HW, there will be 1 gradiance, and a practice problem set on transactions
Conflicting operations (again)

• Two operations on the same data item conflict if at least one of the operations is a write
  • r(X) and w(X) conflict
  • w(X) and r(X) conflict
  • w(X) and w(X) conflict
  • r(X) and r(X) do not conflict
  • r/w(X) and r/w(Y) do not conflict

• Order of conflicting operations matters
  • E.g., if \( T_1.r(A) \) precedes \( T_2.w(A) \), then conceptually, \( T_1 \) should precede \( T_2 \)
Precedence graph

• A node for each transaction
• A directed edge from $T_i$ to $T_j$ if an operation of $T_i$ precedes and conflicts with an operation of $T_j$ in the schedule

**Good:** no cycle

**Bad:** cycle
Conflict-serializable schedule

• A schedule is conflict-serializable iff its precedence graph has no cycles

• A conflict-serializable schedule is equivalent to some serial schedule (and therefore is “good”)
  • In that serial schedule, transactions are executed in the “topological order” of the precedence graph (see next slide)
  • You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions (see next to next slide)
Topological order to find equivalent serial schedule(s)

- List a node only after all its predecessors (nodes having a directed path to this node) are processed.

Equivalent serial schedule(s)

- T1, T2, T3
- T1, T3, T4, T2
- T1, T4, T3, T2

OR

- T1, T2, T3
- T2, T1, T3
- T1, T3, T4, T2
- T1, T4, T3, T2

OR
Swapping adjacent non-conflicting actions to reach an equivalent serial schedule

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r(A)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w(A)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r(B)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w(B)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w(C)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r(C)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$w(B)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r(B)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Good: no cycle

SERIAL
Locking (for Concurrency Control)

- Rules
  - If a transaction wants to **read** an object, it must first request a **shared lock (S mode)** on that object.
  - If a transaction wants to **modify** an object, it must first request an **exclusive lock (X mode)** on that object.
  - Allow one exclusive lock, or multiple shared locks.

<table>
<thead>
<tr>
<th>Mode of lock(s) currently held by other transactions</th>
<th>Mode of the lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>X</td>
<td>No</td>
</tr>
</tbody>
</table>
Basic locking is not enough

Possible schedule under locking

Add 1 to both A and B (preserve A=B)
Read 100
Write 100+1

But still not conflict-serializable!

Multiply both A and B by 2 (preserves A=B)
Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks

\[
T_1 \rightarrow T_2 \rightarrow T_1
\]

2PL guarantees a conflict-serializable schedule

Cannot obtain the lock on B until \( T_1 \) unlocks
Remaining problems of 2PL

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(B)</td>
<td>w(B)</td>
</tr>
</tbody>
</table>

- $T_2$ has read uncommitted data written by $T_1$
- If $T_1$ aborts, then $T_2$ must abort as well
- **Cascading aborts** possible if other transactions have read data written by $T_2$
- **Avoids Cascading Rollback** = Each transaction reads only data written by committed transactions.

- Even worse, what if $T_2$ commits before $T_1$?
  - Schedule is not recoverable if the system crashes right after $T_2$ commits
- **Recoverable** = Each transaction commits after all transactions from which it has read has committed.
Strict 2PL

• Only release locks at commit/abort time
  • A writer will block all other readers until the writer commits or aborts

• Used in many commercial DBMS
  • Oracle is a notable exception
Isolation levels not based on locks?

Snapshot isolation in Oracle

• Based on multiversion concurrency control
  • Used in Oracle, PostgreSQL, MS SQL Server, etc.
  • Intuition: uses a “private snapshot” or “local copy”
  • If no conflict make global otherwise abort

• More efficient than locks, but may lead to aborts

• Other methods: Timestamp-based

Some are covered in the grad db course Compsci 516 if you are interested, see the slides