SQL: Transactions

Introduction to Databases
CompSci 316 Spring 2017
Announcements (Mon., Feb. 26)

• **Project Milestone #1** due tonight
  • Only one report per group is needed
  • Upload on sakai
  • Clearly mention all project members

• No homework deadline for a while
  • work on your project!
SQL features covered so far

• Query
  • SELECT-FROM-WHERE statements
  • Aggregation and grouping, subqueries
  • Set, bag, NULLs
  • Ordering
  • Outerjoins

• Modification
  • INSERT/DELETE/UPDATE

• Constraints
  • Keys, foreign keys, CHECK, Assertion, Triggers

• Views

• Indexes

• Recursion
Next

• Transaction in SQL
  • More later in the course (locking and logging)
  • today

• Programming in SQL
  • next lecture

Might be useful in your project
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 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
- the same program executed multiple times would be considered as different transactions
- DBMS will enforce some Integrity Constraints (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:

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

- Consider a possible interleaving (schedule):

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

- This is OK. But what about:

  T1: \[ A=A+100, \quad B=B-100 \]
  T2: \[ A=1.06*A, \quad B=1.06*B \]

- The DBMS’s view of the second schedule:

  T1: \[ \text{R(A), W(A),} \quad \text{R(B), W(B)} \]
  T2: \[ \text{R(A), W(A), R(B), W(B)} \]
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

<table>
<thead>
<tr>
<th>T1: BEGIN</th>
<th>A=A+100,   B=B-100 END</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: BEGIN</td>
<td>A=1.06<em>A, B=1.06</em>B END</td>
</tr>
</tbody>
</table>
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
  - Often implemented with “locking”

- **Recovery**
  - Due to crashes, there can be partial transactions
  - DBMS needs to ensure that they are not visible to other transactions
  - Often implemented with “logging”

```
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
```
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
  • Because it is often difficult to undo a schema operation

• Many DBMS support an AUTOCOMMIT feature, which automatically commits every single statement
  • You can turn it on/off through the API
  • For PostgreSQL:
    • psql command-line processor turns it on by default
    • You can turn it off at the psql prompt by typing: \set AUTOCOMMIT 'off'
ACID properties

• The database operations in a transaction should follow the following properties (ACID):
  • **Atomic**: Operations of a transaction are executed all-or-nothing, and are never left “half-done”
  • **Consistency**: Assume all database constraints are satisfied at the start of a transaction, they should remain satisfied at the end of the transaction
  • **Isolation**: Transactions must behave as if they were executed in complete isolation from each other
  • **Durability**: If the DBMS crashes after a transaction commits, all effects of the transaction must remain in the database when DBMS comes back up
Atomicity – 1/2

- 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

```
T1: BEGIN A=A+100, B=B-100 END
T2: BEGIN A=1.06*A, B=1.06*B END
```
Atomicity – 2/2

Partial effects of a transaction must be undone when

1. User explicitly aborts the transaction using ROLLBACK
   • E.g., application asks for user confirmation in the last step and issues COMMIT or ROLLBACK depending on the response

2. The DBMS crashes before a transaction commits

3. Any constraint is violated
   • Some systems roll back only this statement and let the transaction continue; others roll back the whole transaction

• How is atomicity achieved?
  • By DBMS
  • Using Logging (to support “undo”)
Consistency – 1/2

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

- 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.
Consistency – 2/2

• Ensuring this property is the responsibility of the user in each transaction

• Consistency of the database is guaranteed by constraints and triggers declared in the database and/or transactions themselves

• Whenever inconsistency arises, abort the statement or transaction, or (with deferred constraint checking or application-enforced constraints) fix the inconsistency within the transaction
Isolation – 1/2

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

- 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
Isolation – 2/2

• Transactions must appear to be executed in a **serial schedule** (with no interleaving operations)

• For performance, DBMS executes transactions using a **serializable schedule**
  • In this schedule, operations from different transactions can interleave and execute concurrently
  • But the schedule is guaranteed to produce the same effects as a serial schedule

• **How is isolation achieved?**
  • Locking, multi-version concurrency control, etc.
Durability – 1/2

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
Durability – 2/2

• DBMS accesses data on stable storage by bringing data into memory

• Effects of committed transactions must survive DBMS crashes

• How is durability achieved?
  • Forcing all changes to disk at the end of every transaction?
    • Too expensive
  • By Logging again
  • now to support “redo”!
  • for atomicity it was “undo”
SQL isolation levels

• Strongest isolation level: **SERIALIZABLE**
  • Mimics “complete isolation”
  • i.e. as if the transactions are executed one by one (serial schedule)
  • the executed schedule is equivalent to such a schedule (therefore is “serializable”)

• 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
  • 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;
  
  ROLLBACK;

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

  COMMIT;
READ COMMITTED

• No dirty reads, but **non-repeatable reads** possible
  • 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;
    ```
  ```sql
  SELECT AVG(pop) 
  FROM User; 
  COMMIT; 
  ```
REPEATABLE READ

• Reads are repeatable, but may see phantoms
• 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**
ANSI isolation levels are lock-based

• **READ UNCOMMITTED**
  - Short-duration locks: lock, access, release immediately

• **READ COMMITTED**
  - Long-duration write locks: do not release write locks until commit

• **REPEATABLE READ**
  - Long-duration locks on all data items accessed

• **SERIALIZABLE**
  - Lock ranges to prevent insertion as well
Isolation levels not based on locks?

Snapshot isolation in Oracle
• Based on multiversion concurrency control
  • Used in Oracle, PostgreSQL, MS SQL Server, etc.
• How it works
  • Transaction $X$ performs its operations on a private snapshot of the database taken at the start of $X$
  • $X$ can commit only if it does not write any data that has been also written by a transaction committed after the start of $X$
• Avoids all ANSI anomalies
• But is NOT equivalent to SERIALIZABLE because of write skew anomaly
Write skew example

• Constraint: combined balance $A + B \geq 0$
• $A = 100, B = 100$
• $T_1$ checks $A + B - 200 \geq 0$, and then proceeds to withdraw 200 from $A$
• $T_2$ checks $A + B - 200 \geq 0$, and then proceeds to withdraw 200 from $B$
• Possible under snapshot isolation because the writes (to $A$ and to $B$) do not conflict
• But $A + B = -200 < 0$ afterwards!
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 safer