Announcement

- Reading assignments for this week
  - “A Critique of ANSI SQL Isolation Levels,” by Berenson et al. in SIGMOD 1995
  - “Weaving Relations for Cache Performance,” by Ailamaki et al. in VLDB 2001
- Recitation session this Friday (February 7)
  - SQL/application programming
  - Help on Homework #1
- Reminder: Homework #1 due in 7 days

Summary of SQL features covered so far

- Query
  - SELECT-FROM-WHERE statements, set and bag operations, table expressions, subqueries, ordering, aggregation and grouping
- Modification
  - INSERT/DELETE/UPDATE
- Constraints
- Triggers
- Views
- Indexes

* Next: transactions and SQL programming
Transactions

A transaction is a sequence of database operations with the following properties (ACID):
- Atomicity: 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

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
  - These statements can see the changes made by earlier statements in this transaction
  - Statements in other concurrently running transactions should not see these changes
- 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
- You can turn on/off a feature called AUTOCOMMIT, which automatically commits every single statement
Atomicity

- Partial effects of a transaction must be undone when
  - User explicitly aborts the transaction using ROLLBACK
  - Application asks for user confirmation in the last step and issues COMMIT or ROLLBACK depending on the response
  - The DBMS crashes before a transaction commits
- Partial effects of a modification statement must be undone when any constraint is violated
  - However, only this statement is rolled back; the transaction continues
- How is atomicity achieved?
  - Logging

Durability

- Effects of committed transactions must survive DBMS crashes
- How is durability achieved?
  - DBMS manipulates data in memory; forcing all changes to disk at the end of every transaction is very expensive
  - Logging

Consistency

- Consistency of the database is guaranteed by constraints and triggers declared in the database and/or transactions themselves
  - When inconsistency arises, abort the statement or transaction, or (with deferred constraint checking or for application-enforced constraints) fix the inconsistency within the transaction
Isolation

- 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.

SQL isolation levels

- Strongest isolation level: SERIALIZABLE
  - Complete isolation
  - SQL default
- 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:

- Example: wrong average
  - -- T1:                      -- T2:
    UPDATE Student
    SET GPA = 3.0
    WHERE SID = 142;
    SELECT AVG(GPA)
    FROM Student;
    ROLLBACK;
    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: -- T2:`
    ```sql
    UPDATE Student
    SET GPA = 3.0
    WHERE SID = 142;
    COMMIT;

    SELECT AVG(GPA)
    FROM Student;
    COMMIT;
    ```

REPEATABLE READ

- Reads are repeatable, but may see phantoms
- Example: different average (still!)
  - `-- T1: -- T2:`
    ```sql
    UPDATE Student
    SET GPA = 3.0
    WHERE SID = 142;
    COMMIT;

    SELECT AVG(GPA)
    FROM Student;
    COMMIT;
    ```

Summary of SQL isolation levels

<table>
<thead>
<tr>
<th>Isolation level/ anomaly</th>
<th>Dirty reads</th>
<th>Non-repeatable reads</th>
<th>Phantom</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
- Criticized recently for being ambiguous and incomplete
  - See reading assignment
SQL Programming

- Pros and cons of SQL
  - Very high-level, possible to optimize
  - Not intended for general-purpose computation
- Solutions
  - Inside: augment SQL with constructs from general-purpose programming languages (e.g., SQL/PSM, Oracle PL/SQL, etc.)
  - Outside: use SQL together with general-purpose programming languages (e.g., JDBC, SQLJ, etc.)

Impedance mismatch and a solution

- SQL operates on a set of records at a time
- Typical low-level general-purpose programming languages operates on one record at a time
- Solution: cursors
  - Open (a table or a result table): position the cursor just before the first row
  - Get next: move the cursor to the next row and return that row
  - Close: clean up and release DBMS resources
- Found in virtually every database language/API (with slightly different syntaxes)
- Some support more cursor positioning and movement options, modification at the current cursor position, etc.

Augmenting SQL: SQL/PSM example

```sql
CREATE FUNCTION SetMaxGPA(IN newMaxGPA FLOAT)
RETURNS INT
-- Enforce newMaxGPA; return number of rows modified.
BEGIN
  DECLARE rowsUpdated INT DEFAULT 0;
  DECLARE thisGPA FLOAT;
  -- A cursor to range over all students:
  DECLARE studentCursor CURSOR FOR
  SELECT GPA FROM Student
  FOR UPDATE;
  -- Set a flag whenever there is a "not found" exception:
  DECLARE noMoreRows INT DEFAULT 0;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
  SET noMoreRows = 1;
  ... (see next slide) ...
  RETURN rowsUpdated;
END
```
SQL/PSM example continued

```sql
-- Fetch the first result row:
OPEN studentCursor;
FETCH FROM studentCursor INTO thisGPA;
-- Loop over all result rows:
WHILE notMoreRows <> 1 DO
  IF thisGPA > newMaxGPA THEN
    -- Enforce newMaxGPA:
    UPDATE Student SET Student.GPA = newMaxGPA
    WHERE CURRENT OF studentCursor;
    -- Update count:
    SET rowsUpdated = rowsUpdated + 1;
  END IF;
  -- Fetch the next result row:
  FETCH FROM studentCursor INTO thisGPA;
END WHILE;
CLOSE studentCursor;
```

Interfacing SQL with another language

- **API approach**
  - SQL commands are sent to the DBMS at runtime
  - Examples: JDBC, ODBC (for C/C++/VB), Perl DBI
  - These API's are all based on the SQL/CLI (Call-Level Interface) standard
- **Embedded SQL approach**
  - SQL commands are embedded in application code
  - A precompiler checks these commands at compile-time and convert them into DBMS-specific API calls
  - Examples: embedded SQL for C/C++, SQLJ (for Java)

Example API: JDBC

```java
// Execute a query and get its results:
ResultSet rs =
  stmt.executeQuery("SELECT SID, name FROM Student");
// Loop through all result rows:
while (rs.next()) {
  // Get column values:
  int sid = rs.getInt(1);
  String name = rs.getString(2);
  // Work on sid and name:
  ...
}
// Close the ResultSet:
rs.close();
```
Some other useful JDBC features

- Prepared statements
  - For every SQL string it gets, the DBMS must perform parsing, semantic analysis, optimization, compilation, and execution
  - Precompile frequently used statement patterns (e.g., "SELECT name FROM Student WHERE SID = ?") into prepared statements
  - Execute prepared statements with actual parameter values
  - The DBMS only needs to validate the parameter values and the compiled execution plan before executing it

- Transaction support
  - Set isolation level for current transaction
  - Turn on/off AUTOCOMMIT (commits every single statement)
  - Commit/rollback current transaction (when AUTOCOMMIT is off)

Example of embedding SQL in C

```c
... /* Declare variables to be "shared" between application and DBMS: */
EXEC SQL BEGIN DECLARE SECTION;
int thisSID; float thisGPA;
EXEC SQL END DECLARE SECTION;
/* Declare a cursor: */
EXEC SQL DECLARE StudentCursor CURSOR FOR
  SELECT SID, GPA FROM Student;
EXEC SQL OPEN StudentCursor; /* Open the cursor */
EXEC SQL WHENEVER NOT FOUND DO break; /* Specify exit condition */
/* Loop through result rows: */
while (1) {
  /* Get column values for the current row: */
  EXEC SQL FETCH StudentCursor INTO :thisSID, :thisGPA;
}
EXEC SQL CLOSE CPS196Student; /* Close the cursor */
... 
```

Pros and cons of embedded SQL

- Pros

- Cons
Pros and cons of augmenting SQL

- Pros
- Cons