

# SQL: Part IV

CPS 216  
Advanced Database Systems

## Announcements

- ❖ 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



## READ COMMITTED

13

- ❖ No dirty reads, but non-repeatable reads possible
  - Reading the same data item twice can produce different results

- ❖ Example: different averages

```
-- T1:
UPDATE Student
SET GPA = 3.0
WHERE SID = 142;
COMMIT;

-- T2:
SELECT AVG(GPA)
FROM Student;

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

## REPEATABLE READ

14

- ❖ Reads are repeatable, but may see phantoms

- ❖ Example: different average (still!)

```
-- T1:
INSERT INTO Student
VALUES(789, 'Nelson', 10, 1.0);
COMMIT;

-- T2:
SELECT AVG(GPA)
FROM Student;

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

## Summary of SQL isolation levels

15

| Isolation level/anomaly | Dirty reads | Non-repeatable reads | Phantoms   |
|-------------------------|-------------|----------------------|------------|
| READ UNCOMMITTED        | Possible    | Possible             | Possible   |
| READ COMMITTED          | Impossible  | Possible             | Possible   |
| REPEATABLE READ         | Impossible  | Impossible           | Possible   |
| SERIALIZABLE            | Impossible  | Impossible           | Impossible |

- ❖ Syntax: At the beginning of a transaction,  
SET TRANSACTION ISOLATION LEVEL *isolation\_level*  
{READ ONLY|READ WRITE};
  - READ UNCOMMITTED can only be READ ONLY (why?)
- ☞ Criticized recently for being ambiguous and incomplete
  - See reading assignment

## SQL Programming

16

- ❖ 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

17

- ❖ 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; raise a flag if there is no more next 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

18

```
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

19

```
-- Fetch the first result row:
OPEN studentCursor;
FETCH FROM studentCursor INTO thisGPA;
-- Loop over all result rows:
WHILE noMoreRows <> 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

20

### ❖ 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

21

```
...
// 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

22

### ❖ 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

23

```
...
/* 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 StudentCursor; /* Close the cursor */
...
```

## Pros and cons of embedded SQL

24

### ❖ Pros

- More compile-time checking (syntax, type, schema, ...)
- Code could be more efficient (if the embedded SQL statements do not need to be checked and recompiled at run-time)

### ❖ Cons

- DBMS-specific
  - Vendors have different precompilers which translate code into different native API's
  - Application executable is not portable (although code is)
  - Application cannot talk to different DBMS at the same time

## Pros and cons of augmenting SQL

### ❖ Pros

- More sophisticated stored procedures and triggers
- More application logic can be pushed closer to data

### ❖ Cons

- Already too many programming languages
- SQL is already too big
- General-purpose programming constructs complicate optimization make it impossible to tell if code running inside the DBMS is safe
- At some point, one must recognize that SQL and the DBMS engine are not for everything!