Ordering

ORDER BY

- `SELECT [DISTINCT] ...`
  `FROM ... WHERE ... GROUP BY ... HAVING ...`
  `ORDER BY` *output_column* `[ASC | DESC], ...;`
- `ASC` = ascending, `DESC` = descending
- Operational semantics
  - After `SELECT` list has been computed and optional duplicate elimination has been carried out, sort the output according to `ORDER BY` specification

ORDER BY example

- List all students, sort them by GPA (descending) and then name (ascending)
  - ```
    SELECT SID, name, age, GPA
    FROM Student
    ORDER BY GPA DESC, name;
    ```
  - `ASC` is the default option
  - Strictly speaking, only output columns can appear in `ORDER BY` clause (although some DBMS support more)
  - Can use sequence numbers of output columns instead
    `ORDER BY 4 DESC, 2;`

Triggers

"Active" data

- Constraint enforcement: When a transaction violates a constraint, abort the transaction or try to "fix" the data
  - Example: enforcing referential integrity constraints
  - Generalize to arbitrary constraints?
- Data monitoring: When something happens to the data, automatically execute some action
  - Example: When price rises above $20 per share, sell
  - Example: When enrollment is at the limit and more students try to register, email the instructor

Triggers

- A trigger is an event-condition-action rule
  - When event occurs, test condition; if condition is satisfied, execute action
- Example:
  - Event: whenever there comes a new student…
  - Condition: with GPA higher than 3.0…
  - Action: then make him/her take CPS216!
    ```

Trigger example

```
CREATE TRIGGER CPS216AutoRecruit
  AFTER INSERT ON Student
  REFERENCING NEW ROW AS newStudent
  FOR EACH ROW
  WHEN (newStudent.GPA > 3.0)
  INSERT INTO Enroll
      VALUES(newStudent.SID, 'CPS216');
```

Event: INSERT ON Student

Condition: (newStudent.GPA > 3.0)

Action: INSERT INTO Enroll VALUES(newStudent.SID, 'CPS216');

Trigger options

- Possible events include:
  - `INSERT ON` *table*
  - `DELETE ON` *table*
  - `UPDATE [OF` *column*`] ON` *table*
- Trigger can be activated:
  - `FOR EACH ROW` modified
  - `FOR EACH STATEMENT` that performs modification
- Action can be executed:
  - `AFTER` or `BEFORE` the triggering event

Transition variables

- `OLD ROW`: the modified row before the triggering event
- `NEW ROW`: the modified row after the triggering event
- `OLD TABLE`: a hypothetical read-only table containing all modified rows before the triggering event
- `NEW TABLE`: a hypothetical table containing all modified rows after the triggering event
- Not all of them make sense all the time, e.g.
  - `AFTER INSERT` statement-level triggers
    - Can use only `NEW TABLE`
  - `BEFORE DELETE` row-level triggers
    - Can use only `OLD ROW`
  - etc.

Statement-level trigger example

```
CREATE TRIGGER CPS216AutoRecruit
  AFTER INSERT ON Student
  REFERENCING NEW TABLE AS newStudents
  FOR EACH STATEMENT
  INSERT INTO Enroll
  (SELECT SID, 'CPS216'
   FROM newStudents
   WHERE GPA > 3.0);
```


`BEFORE` trigger example

- Never give faculty more than 50% raise in one update

```
CREATE TRIGGER NotTooGreedy
BEFORE UPDATE OF salary ON Faculty
REFERENCING OLD ROW AS o, NEW ROW AS n
FOR EACH ROW
WHEN (n.salary > 1.5 * o.salary)
SET n.salary = 1.5 * o.salary;
```

- ☞ `BEFORE` triggers are often used to "condition" data
- ☞ Another option is to raise an error in the trigger body to abort the transaction that caused the trigger to fire

Statement- vs. row-level triggers

Why are both needed?

- Certain triggers are only possible at statement level
  - If the average GPA of students inserted by this statement exceeds 3.0, do …
- Simple row-level triggers are easier to implement and may be more efficient
  - Statement-level triggers require significant amount of state to be maintained in `OLD TABLE` and `NEW TABLE`
  - However, a row-level trigger does get fired for each row, so complex row-level triggers may be inefficient for statements that generate lots of modifications

System issues

- Recursive firing of triggers
  - Action of one trigger causes another trigger to fire
  - Can get into an infinite loop
    - Some DBMS restrict trigger actions
    - Most DBMS set a maximum level of recursion (16 in DB2)
- Interaction with constraints (very tricky to get right!)
  - When do we check if a triggering event violates constraints?
    - After a `BEFORE` trigger (so the trigger can fix a potential violation)
    - Before an `AFTER` trigger
  - `AFTER` triggers also see the effects of, say, cascaded deletes caused by referential integrity constraint violations

(Based on DB2; other DBMS may implement a different policy!)

Transactions

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: What if the transaction that wrote the dirty data eventually aborts?
- 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:
                                SELECT AVG(GPA)
                                FROM Student;
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:
                                    SELECT AVG(GPA)
                                    FROM Student;
INSERT INTO Student
VALUES(789, 'Nelson', 10, 1.0);
COMMIT;
                                    SELECT AVG(GPA)
                                    FROM Student;
                                    COMMIT;
```


Summary of SQL isolation levels

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

Application Programming

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; 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

```
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 CUSOR 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

```
-- 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

- 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

```
…
// 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

```
…
/* 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
  - More compile-time checking (syntax, type, schema, …)
  - Code could be more efficient (if the embedded SQL statements do not need to 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!