CS425 – Fall 2013
Boris Glavic
Chapter 6: Advanced SQL

modified from:

Database System Concepts, 6th Ed.

Chapter 6: Advanced SQL

- Accessing SQL From a Programming Language
  - Dynamic SQL
    - JDBC and ODBC
  - Embedded SQL
- Functions and Procedural Constructs
- Triggers

Textbook: Chapter 5

Accessing SQL From a Programming Language

JDBC and ODBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
  - Connect with the database server
  - Send SQL commands to the database server
  - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
  - Other API's such as ADO.NET sit on top of ODBC
- JDBC (Java Database Connectivity) works with Java

Native APIs

- Most DBMS also define DBMS specific APIs

- Oracle: OCI
- Postgres: libpg

…

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver"); // load driver
        Connection conn = DriverManager.getConnection(  // connect to server
              "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement(); // create Statement object
            … Do Actual Work ….
        stmt.close(); // close Statement and release resources
        conn.close(); // close Connection and release resources
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle); // handle exceptions
    }
}
```

JDBC Code (Cont.)

- Update to database

```
try {
     stmt.executeUpdate(
          "insert into instructor values(' 77987' , ' Kim' , ' Physics' ,
98000)");
} catch (SQLException sqle)
{
    System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
       ResultSet rset = stmt.executeQuery(
                              "select dept_name, avg (salary)
                               from instructor
                               group by dept_name");
      while (rset.next()) {
             System.out.println(rset.getString("dept_name") + " " +
                                     rset.getFloat(2));
      }
```

JDBC Code Details

- Result stores the current row position in the result
  - Pointing before the first row after executing the statement
  - **.next()** moves to the next tuple
    - Returns false if no more tuples
- Getting result fields:
  - **rs.getString("dept_name") and rs.getString(1) equivalent if dept_name is the first attribute in select result.**
- Dealing with Null values
  - **int a = rs.getInt("a");**

    if (rs.wasNull()) Systems.out.println("Got null value");

Prepared Statement

- PreparedStatement pStmt = conn.prepareStatement(
                                    "insert into instructor values(?,?,?,?)");
  pStmt.setString(1, "88877");   pStmt.setString(2, "Perry");
  pStmt.setString(3, "Finance");   pStmt.setInt(4, 125000);
  pStmt.executeUpdate();
  pStmt.setString(1, "88878");
  pStmt.executeUpdate();
- For queries, use pStmt.executeQuery(), which returns a ResultSet
- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - **NEVER create a query by concatenating strings which you get as inputs**
  - "insert into instructor values(' " + ID + " ' , ' " + name + " ' , " +
                                        " ' + dept name + " ' , " ' balance +
    ")"
  - What if name is "D' Souza"?

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = ' " + name + "' "
- Suppose the user, instead of entering a name, enters:
  - X' or ' Y' = ' Y
- then the resulting statement becomes:
  - "select * from instructor where name = ' " + "X' or ' Y' = ' Y" + "' "
  - which is:
    - select * from instructor where name = ' X' or ' Y' = ' Y'
  - User could have even used
    - X' ; update instructor set salary = salary + 10000; --
- Prepared statement internally uses:
  "select * from instructor where name = ' X\' or \' Y\' = \' Y'
  - **Always use prepared statements, with user inputs as parameters**

Metadata Features

- ResultSet metadata
- E.g., after executing query to get a ResultSet rs:
  - ResultSetMetaData rsmd = rs.getMetaData();

```
    for(int i = 1; i <= rsmd.getColumnCount(); i++) {
        System.out.println(rsmd.getColumnName(i));
        System.out.println(rsmd.getColumnTypeName(i));
    }
```

- How is this useful?

Metadata (Cont)

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();

```
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
while( rs.next()) {
    System.out.println(rs.getString("COLUMN_NAME"),
                               rs.getString("TYPE_NAME");
}
```

- And where is this useful?

Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); or
  - conn.rollback();
- conn.setAutoCommit(true) turns on automatic commit.

Other JDBC Features

- Calling functions and procedures
  - CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");
  - CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");
- Handling large object types
  - getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively
  - get data from these objects by getBytes()
  - associate an open stream with Java Blob or Clob object to update large objects
    - blob.setBlob(int parameterIndex, InputStream inputStream).

SQLJ

- JDBC is overly dynamic, errors cannot be caught by compiler
- SQLJ: embedded SQL in Java
  - #sql iterator deptInfoIter (String dept name, int avgSal);

```
deptInfoIter iter = null;
#sql iter = { select dept_name, avg(salary) from instructor
            group by dept name };
while (iter.next()) {
    String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
}
iter.close();
```

ODBC

- Open DataBase Connectivity(ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
- Was defined originally for Basic and C, versions available for many languages.

ODBC (Cont.)

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.

ODBC Code

- int ODBCexample()

```
{
   RETCODE error;
   HENV    env;    /* environment */
   HDBC    conn;  /* database connection */
   SQLAllocEnv(&env);
   SQLAllocConnect(env, &conn);
   SQLConnect(conn, “db.yale.edu", SQL_NTS, "avi", SQL_NTS,
     "avipasswd", SQL_NTS);
   { .... Do actual work … }

   SQLDisconnect(conn);
   SQLFreeConnect(conn);
   SQLFreeEnv(env);
}
```

ODBC Code (Cont.)

- Program sends SQL commands to database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
  - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
  - Arguments to SQLBindCol()
    - ODBC stmt variable, attribute position in query result
    - The type conversion from SQL to C.
    - The address of the variable.
    - For variable-length types like character arrays,
      - The maximum length of the variable
      - Location to store actual length when a tuple is fetched.
      - Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.

ODBC Code (Cont.)

- Main body of program

```
 char deptname[80];
float salary;
int lenOut1, lenOut2;
HSTMT stmt;
char * sqlquery = "select dept_name, sum (salary)
                   from instructor
                   group by dept_name";
SQLAllocStmt(conn, &stmt);
error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
if (error == SQL SUCCESS) {
     SQLBindCol(stmt, 1, SQL_C_CHAR, deptname , 80, &lenOut1);
     SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0 , &lenOut2);
     while (SQLFetch(stmt) == SQL_SUCCESS) {
          printf (" %s %g\n", deptname, salary);
     }
}
SQLFreeStmt(stmt, SQL_DROP);
```

ODBC Prepared Statements

- **Prepared Statement**
  - SQL statement prepared: compiled at the database
  - Can have placeholders: E.g. insert into account values(?,?,?)
  - Repeatedly executed with actual values for the placeholders
- To prepare a statement
  SQLPrepare(stmt, <SQL String>);
- To bind parameters
  SQLBindParameter(stmt, <parameter#>,
  … type information and value omitted for simplicity..)
- To execute the statement
  retcode = SQLExecute(stmt);
- To avoid SQL injection security risk, do not create SQL strings directly using user input; instead use prepared statements to bind user inputs

More ODBC Features

- **Metadata features**
  - finding all the relations in the database and
  - finding the names and types of columns of a query result or a relation in the database.
- By default, each SQL statement is treated as a separate transaction that is committed automatically.
  - Can turn off automatic commit on a connection
    - SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)}
  - Transactions must then be committed or rolled back explicitly by
    - SQLTransact(conn, SQL_COMMIT) or
    - SQLTransact(conn, SQL_ROLLBACK)

ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
  - Core
  - Level 1 requires support for metadata querying
  - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.

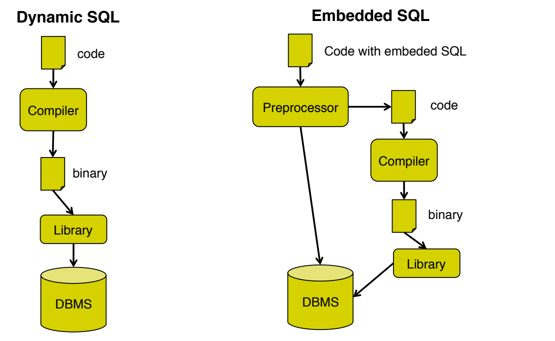
ADO.NET

- API designed for Visual Basic .NET and C#, providing database access facilities similar to JDBC/ODBC
  - Partial example of ADO.NET code in C#

```
using System, System.Data, System.Data.SqlClient;
SqlConnection conn = new SqlConnection(
        "Data Source=<IPaddr>, Initial Catalog=<Catalog>");
conn.Open();
SqlCommand cmd = new SqlCommand("select * from students",
                                   conn);
SqlDataReader rdr = cmd.ExecuteReader();
while(rdr.Read()) {
   Console.WriteLine(rdr[0], rdr[1]); /* Prints result attributes 1 & 2 */
}
rdr.Close(); conn.Close();
```

- Can also access non-relational data sources such as
  - OLE-DB, XML data, Entity framework

Dynamic vs. Embedded SQL



Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/I.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement > END_EXEC

  Note: this varies by language (for example, the Java embedding uses # SQL { …. };)

Example Query

- From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount.
- Specify the query in SQL and declare a *cursor* for it

  EXEC SQL

  declare *c* **cursor for**
  select *ID, name*
  from *student*
  where tot_cred > *:credit_amount*

  END_EXEC

Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated

  EXEC SQL **open** *c* END_EXEC

- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

  EXEC SQL **fetch** *c* **into** *:si, :sn* END_EXEC

  Repeated calls to **fetch** get successive tuples in the query result
- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  EXEC SQL **close** *c* END_EXEC

  Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

  declare *c* **cursor for**
    select *
    from *instructor*
    where *dept_name* = 'Music'
  for update

- To update tuple at the current location of cursor *c*

  update *instructor*
  set *salary* = *salary* + 100
  where current of *c*

Procedural Constructs in SQL

Procedural Extensions and Stored Procedures

- SQL provides a **module** language
  - Permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
- Stored Procedures
  - Can store procedures in the database
  - then execute them using the **call** statement
  - permit external applications to operate on the database without knowing about internal details
- Object-oriented aspects of these features are covered in Chapter 22 (Object Based Databases) in the textbook

Why have procedural extensions?

- Shipping data between a database server and application program (e.g., through network connection) is costly
- Converting data from the database internal format into a format understood by the application programming language is costly
- Example:
  - Use Java to retrieve all users and their friend-relationships from a friends relation representing a world-wide social network with 10,000,000 users
  - Compute the transitive closure
    - All pairs of users connects through a path of friend relationships. E.g., (Peter, Magret) if Peter is a friend of Walter who is a friend of Magret
  - Return pairs of users from Chicago – say 4000 pairs
  - 1) cannot be expressed (efficiently) as SQL query, 2) result is small
    - -> save by executing this on the DB server

Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language.
  - Functions are particularly useful with specialized data types such as images and geometric objects.
    - Example: functions to check if polygons overlap, or to compare images for similarity.
  - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))
 returns integer
 begin
     declare d_count integer;
     select count ( * ) into d_count
     from instructor
     where instructor.dept_name = dept_name;
     return d_count;
  end
```

- Find the department name and budget of all departments with more that 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name ) > 1
```

Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```
create function instructors_of (dept_name char(20)
        returns table (  ID varchar(5),
                         name varchar(20),
                         dept_name varchar(20),
                         salary numeric(8,2))
return table
        (select ID, name, dept_name, salary
         from instructor
         where instructor.dept_name = instructors_of.dept_name)
```

- Usage

```
        select *
        from table (instructors_of ( 'Music' ))
```

SQL Procedures

- The *dept_count* function could instead be written as procedure:

  create procedure *dept_count_proc* (**in** *dept_name* **varchar**(20),
                                              out *d_count* **integer)**
  begin
    select count(*) **into** *d_count*
    from *instructor*
    where *instructor.dept_name = dept_count_proc.dept_name*
  end
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

  declare *d_count* **integer**;
  call *dept_count_proc*('Physics', *d_count*);

  Procedures and functions can be invoked also from dynamic SQL
- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ

Procedural Constructs

- Warning: most database systems implement their own variant of the standard syntax below
  - read your system manual to see what works on your system
- Compound statement: **begin … end,**
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements
- **While** and **repeat** statements :

```
declare n integer default 0;
while n < 10 do
    set n = n + 1
end while

repeat
    set n = n  – 1
until n = 0
end repeat
```

Procedural Constructs (Cont.)

- **For** loop
  - Permits iteration over all results of a query
  - Example:

```
declare n  integer default 0;
for r  as
        select budget from department
         where dept_name = 'Music'
do
        set n = n - r.budget
end for
```

Procedural Constructs (cont.)

- Conditional statements (**if-then-else**)
  SQL:1999 also supports a **case** statement similar to C case statement
- Example procedure: registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - See book for details
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
…
 ..  signal out_of_classroom_seats
end
```

  - The handler here is **exit** -- causes enclosing **begin..end** to be exited
  - Other actions possible on exception

External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),
                                            out count integer)
language C
external name ' /usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power.
- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space.
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data
  - There are alternatives, which give good security at the cost of potentially worse performance.
  - Direct execution in the database system's space is used when efficiency is more important than security.

Security with External Language Routines

- To deal with security problems
  - Use **sandbox** techniques
    - E.g., use a safe language like Java, which cannot be used to access/damage other parts of the database code.
  - Or, run external language functions/procedures in a separate process, with no access to the database process' memory.
    - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space.

Triggers

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Trigger Example

- E.g. *time_slot_id* is not a primary key of *timeslot,* so we cannot create a foreign key constraint from *section* to *timeslot.*
- Alternative: use triggers on *section* and *timeslot* to enforce integrity constraints

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
            select time_slot_id
            from time_slot)) /* time_slot_id not present in time_slot */
begin
    rollback
end;
```

Trigger Example Cont.

```
create trigger timeslot_check2 after delete on timeslot
    referencing old row as orow
    for each row
    when (orow.time_slot_id not in (
            select time_slot_id
            from time_slot)
            /* last tuple for time slot id deleted from time slot */
        and orow.time_slot_id in (
            select time_slot_id
            from section)) /* and time_slot_id still referenced from section*/
    begin
      rollback
    end;
```

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - **E.g., after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blank grades to null.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
 begin atomic
      set nrow.grade = null;
 end;
```

Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> 'F' **and** *nrow.grade* **is not null**
  and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
  begin atomic
  update *student*
  set *tot_cred*= *tot_cred* +
  (**select** *credits*
  from *course*
  where *course.course_id*= *nrow.course_id*)
  where *student.id* = *nrow.id*;
  end;

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger

When Not To Use Triggers

- Risk of unintended execution of triggers, for example, when
  - loading data from a backup copy
  - replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

  with recursive *rec_prereq*(*course_id*, *prereq_id*) **as** (
  select *course_id*, *prereq_id*
  from *prereq*
  union
  select *rec_prereq.course_id***,** *prereq.prereq_id*,
  from *rec_rereq*, *prereq*
  where *rec_prereq.prereq_id* = *prereq.course_id*
)
  select ∗
  from *rec_prereq*;

  This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be **monotonic**. That is,
  - if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

Another Recursion Example

- Given relation
  manager(*employee_name, manager_name*)
- Find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)

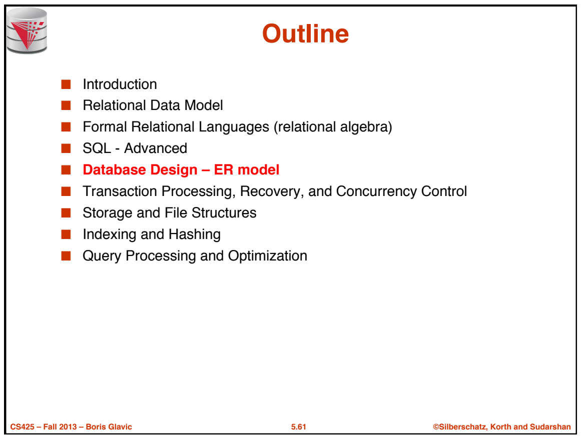
```
with recursive empl (employee_name, manager_name ) as (
        select employee_name, manager_name
        from   manager
    union
        select manager.employee_name, empl.manager_name
        from   manager, empl
        where manager.manager_name = empl.employe_name)
select *
from   empl
```

This example view, *empl,* is the *transitive closure* of the *manager* relation

Recap

- Programming Language Interfaces for Databases
  - Dynamic SQL (e.g., JDBC, ODBC)
  - Embedded SQL
  - SQL Injection
- Procedural Extensions of SQL
  - Functions and Procedures
- External Functions/Procedures
  - Written in programming language (e.g., C)
- Triggers
  - Events (insert, …)
  - Conditions (WHEN)
  - per statement / per row
  - Accessing old/new table/row versions
- Recursive Queries

End of Chapter



Outline

- Introduction
- Relational Data Model
- Formal Relational Languages (relational algebra)
- SQL - Advanced
- **Database Design – ER model**
- Transaction Processing, Recovery, and Concurrency Control
- Storage and File Structures
- Indexing and Hashing
- Query Processing and Optimization