Chapter 5: Advanced SQL

Outline

- Accessing SQL From a Programming Language
- Functions and Procedures

- *Triggers
- **Recursive Queries
- **Advanced Aggregation Features

Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language for at least two reasons

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

Accessing SQL from a Programming Language (Cont.)

There are two approaches to accessing database from a general-purpose programming language

- API（Application Program Interface）- - A general-purpose program can connect to and communicate with a database server using a collection of functions.
- Embedded SQL -- provides a means by which a program can interact with a database server.
  - The SQL statements are translated at compile time into function calls.
  - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

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#
- JDBC (Java Database Connectivity) works with Java
- Embedded SQL in C
- SQLJ - embedded SQL in Java
- JPA(Java Persistence API) - OR mapping of Java

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 {
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
          … Do Actual Work ….
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

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

- Getting result fields:
  - rset.getString("dept_name") and rset.getString(1) equivalent if dept_name is the first argument of select result.
- Dealing with Null values
  - int a = rset.getInt("a");

    if (rset.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();

- 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 + " ', " + salary + ")"
  - What if name is “D’Souza”?

SQL Injection(SQL 注入)

- 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 string of the 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;

    then →

    - select * from instructor where name = 'X';

      update instructor set salary = salary + 10000;
- **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));
}
```

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");
}
```

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).

JDBC Resources

- JDBC Basics Tutorial
  - https://docs.oracle.com/javase/tutorial/jdbc/index.html

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) as avgSal 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();
```

SQLJ Resources

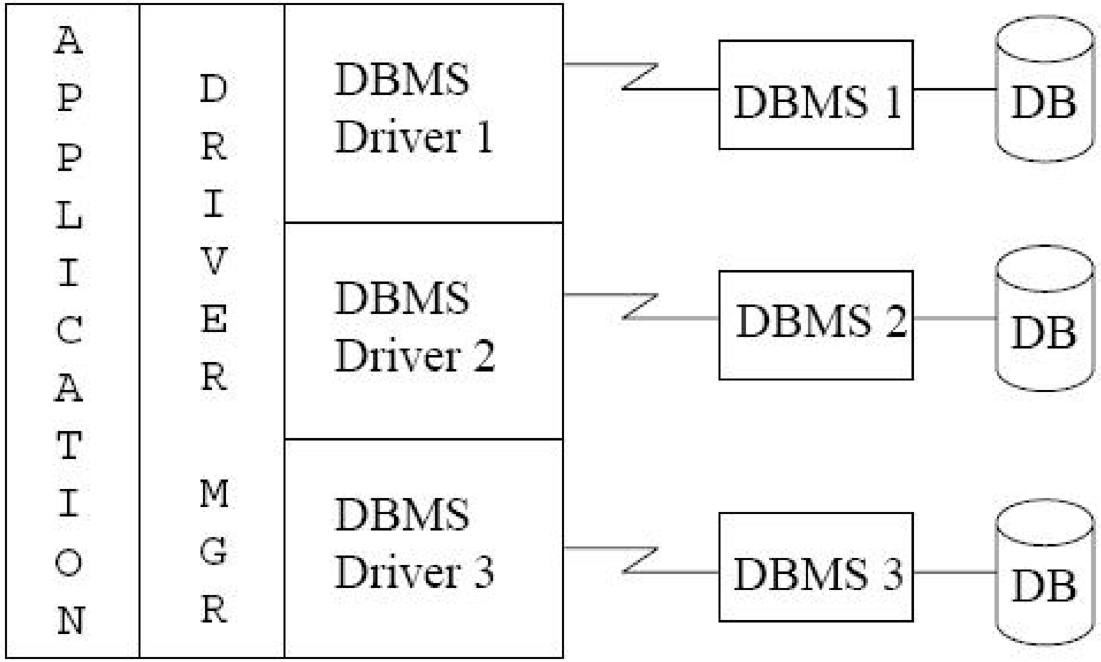
- SQLJ Developer's Guide
- https://docs.oracle.com/en/database/oracle/oracle-database/20/jsqlj/toc.htm

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.



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);

SQL injection（SQL注入）

- An SQL injection example in Java:
- “select * from instructor where name= ’ ” + **name** + ” ’ ”
- malicious hacker enters **name** : X’ or ’Y’ =’Y

  →

  select * from instructor where name= ’ X’ or ’Y’ =’Y ’
- In the resulting SQL statement, the **where** clause is always true and the entire instructor relation is returned.
- Use of prepared statement would prevent this problem because the input string would have escape characters(转义符) inserted, so the resulting query becomes:
- select * from instructor where name= ’ X\’ or \ ’Y\’ =\’Y ’
- This query is harmless and returns the empty relation.
- 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.

ODBC Resources

- MySQL Connector/ODBC Developer Guide
- https://docs.oracle.com/cd/E17952_01/connector-odbc-en/index.html

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,
- 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/1.
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement >;

  Note: this varies by language:

  - In some languages, like COBOL, the semicolon is replaced with END-EXEC
  - In Java embedding uses # SQL { …. };

Embedded SQL (Cont.)

- Before executing any SQL statements, the program must first connect to the database. This is done using:

  EXEC-SQL **connect to** *server* **user** *user-name* **using** *password*;

  Here, *server* identifies the server to which a connection is to be established.

- Variables of the host language can be used within embedded SQL statements. They are preceded by a colon (:) to distinguish from SQL variables (e.g., :*credit_amount*)

- Host Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.

  EXEC-SQL BEGIN DECLARE SECTION

  int *credit-amount* ;

  EXEC-SQL END DECLARE SECTION;

Embedded SQL (Cont.)

- To write an embedded SQL query, we use the

  declare *c* **cursor for <SQL query>**

  statement. The variable *c* is used to identify the query
- Example:
  - 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 in the host langue
  - Specify the query in SQL as follows:

    EXEC SQL

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

Embedded SQL (Cont.)

- The **open** statement for our example is as follows:

  EXEC SQL **open** *c* ;

  This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.

- 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* ;

  Repeated calls to fetch get successive tuples in the query result

Embedded SQL (Cont.)

- 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* ;

Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)
- Can update tuples fetched by cursor by declaring that the cursor is for update

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

- We then iterate through the tuples by performing **fetch** operations on the cursor , and after fetching each tuple we execute the following code:

```
update instructor
set salary = salary + 1000
where current of c
```

Embedded SQL Resources

- Oracle Database Programmer's Guide - Embedded SQL
- https://docs.oracle.com/en/database/oracle/oracle-database/20/lnpcc/embedded-SQL.html#GUID-C671CABF-202A-4503-A16B-DC78D3F1AB13

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

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

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

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**)

  if *boolean expression*
      then *statement or compound statement*
  elseif *boolean expression*
      then *statement or compound statement*
  else *statement or compound statement*
  end if

- SQL:1999 also supports a **case** statement similar to C case statement

Example procedure

```
create function registerStudent(
                in s_id varchar(5),
                in s_courseid varchar (8),
                in s_secid varchar (8),
                in s_semester varchar (6),
                in s_year numeric (4,0),
                out errorMsg varchar(100)
returns integer
begin
     declare currEnrol int;
     select count(*) into currEnrol
          from takes
          where course_id = s_courseid and sec_id = s_secid
               and semester = s_semester and year = s_year;
     declare limit int;
     select capacity into limit
          from classroom natural join section
          where course_id = s_courseid and sec_id = s_secid
               and semester = s_semester and year = s_year;
     if (currEnrol < limit)
          begin
               insert into takes values
                    (s_id, s_courseid, s_secid, s_semester, s_year, null);
               return(0);
          end
     - - Otherwise, section capacity limit already reached
     set errorMsg = 'Enrollment limit reached for course ' || s_courseid
          || ' section ' || s_secid;
     return(-1);
end;
```

– – Registers a student after ensuring
– –   classroom capacity is not exceeded.
– – Returns 0 on success, and -1 if
– –   capacity is exceeded.

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
    - that is 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.
- Trigger - ECA rule
  - E: Event （**insert**, **delete** , **update**）
  - C: Condition
  - A: Action
- 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.

Trigger Example

- account_log(account, amount, datetime)

- **create trigger** *account_trigger* **after update of** *account* **on** *balance*
  referencing new row as *nrow*

  referencing old row as *orow*

  for each row
  when *nrow.balance* - *orow.balance* > =200000 or

  orow.balance -nrow.balance >=50000
  begin
  insert into *account_log* **values** *(nrow.account-number,*

  nrow.balance-orow.balance , current_time())
  end

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

Statement Level Triggers

- **create trigger** *grade_trigger* **after update of** *takes* **on** *grade*

  referencing new table as *new_table*

  for each statement
  when exists(select avg(grade)

  from new_table

  group by course_id, sec_id, semester, year

  having avg(grade)< 60)
  begin
  rollback

  end

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_prereq*, *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 prerequisites.
    - 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 on Page.189

The Power of Recursion

```
create function findAllPrereqs(cid varchar(8))
    -- Finds all courses that are prerequisite (directly or indirectly) for cid
returns table (course_id varchar(8))
    -- The relation prereq(course_id, prereq_id) specifies which course is
    -- directly a prerequisite for another course.
begin
    create temporary table c_prereq (course_id varchar(8));
        -- table c_prereq stores the set of courses to be returned
    create temporary table new_c_prereq (course_id varchar(8));
        -- table new_c_prereq contains courses found in the previous iteration
    create temporary table temp (course_id varchar(8));
        -- table temp is used to store intermediate results
    insert into new_c_prereq
        select prereq_id
        from prereq
        where course_id = cid;
    repeat
        insert into c_prereq
            select course_id
            from new_c_prereq;

        insert into temp
            (select prereq.course_id
                from new_c_prereq, prereq
                where new_c_prereq.course_id = prereq.prereq_id
            )
            except (
                select course_id
                from c_prereq
            );
        delete from new_c_prereq;
        insert into new_c_prereq
            select *
            from temp;
        delete from temp;

    until not exists (select * from new_c_prereq)
    end repeat;
    return table c_prereq;
end
```

Advanced Aggregation Features

Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation

  student_grades(ID, GPA)

  giving the grade-point average of each student
- Find the rank of each student.

  select *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s_rank*
  from *student_grades*
- An extra **order by** clause is needed to get them in sorted order

  select *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s_rank*
  from *student_grades*
  order by *s_rank*
- Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  - **dense_rank** does not leave gaps, so next dense rank would be 2

Ranking

- Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

select *ID*, (1 + (**select count**(*)
        from *student_grades B*
        where *B*.*GPA* > *A*.*GPA*)) **as** *s_rank*
from *student_grades A*
order by *s_rank*;

Ranking (Cont.)

- Ranking can be done within partition of the data.
- "Find the rank of students within each department."

  select *ID*, *dept_name*,
  rank () **over** (**partition by** *dept_name* **order by** *GPA* **desc**)
  as *dept_rank*
  from *dept_grades*
  order by *dept_name*, *dept_rank*;

- Multiple **rank** clauses can occur in a single **select** clause.
- Ranking is done *after* applying **group by** clause/aggregation
- Can be used to find top-n results
  - More general than the **limit** *n* clause supported by many databases, since it allows top-n within each partition

Ranking (Cont.)

- Other ranking functions:
  - **percent_rank** (within partition, if partitioning is done)
  - **cume_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  - **row_number** (non-deterministic in presence of duplicates)
- SQL:1999 permits the user to specify **nulls first** or **nulls last**

select *ID*,
  rank ( ) **over** (**order by** *GPA* **desc nulls last**) **as** *s_rank*
from *student_grades*

Ranking (Cont.)

- For a given constant *n*, the ranking the function *ntile*(*n*) takes the tuples in each partition in the specified order, and divides them into *n* buckets with equal numbers of tuples.
- E.g.,

  select *ID*, **ntile**(4) **over** (**order by** *GPA* **desc**) **as** *quartile*
  from *student_grades;*

Windowing

- Used to smooth out random variations.
- E.g., **moving average**: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”
- **Window specification** in SQL:
  - Given relation *sales(date, value)*

    select *date*, ***sum***(*value*) **over**
    (**order by** *date* **between rows** 1 **preceding and** 1 **following**)
    from *sales*

Windowing

- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **range between** 10 **preceding and current row**
    - All rows with values between current row value –10 to current value
  - **range interval** 10 **day preceding**
    - Not including current row

Windowing (Cont.)

- Can do windowing within partitions
- E.g., Given a relation *transaction* (*account_number, date_time, value*), where value is positive for a deposit and negative for a withdrawal
  - “Find total balance of each account after each transaction on the account”

select *account_number, date_time*,
    sum (*value*) **over**
        (**partition by** *account_number*
        order by *date_time*
        rows unbounded preceding)
    as *balance*
from *transaction*
order by *account_number, date_time*

OLAP**

Data Analysis and OLAP

- **Online Analytical Processing (OLAP)**
  - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes are called **multidimensional data**.
  - **Measure attributes**
    - measure some value
    - can be aggregated upon
    - e.g., the attribute *number* of the *sales* relation
  - **Dimension attributes**
    - define the dimensions on which measure attributes (or aggregates thereof) are viewed
    - e.g., attributes *item_name, color,* and *size* of the *sales* relation

Example sales relation

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6
...	...	...	...
...	...	...	...

Cross Tabulation of *sales* by *item_name* and *color*

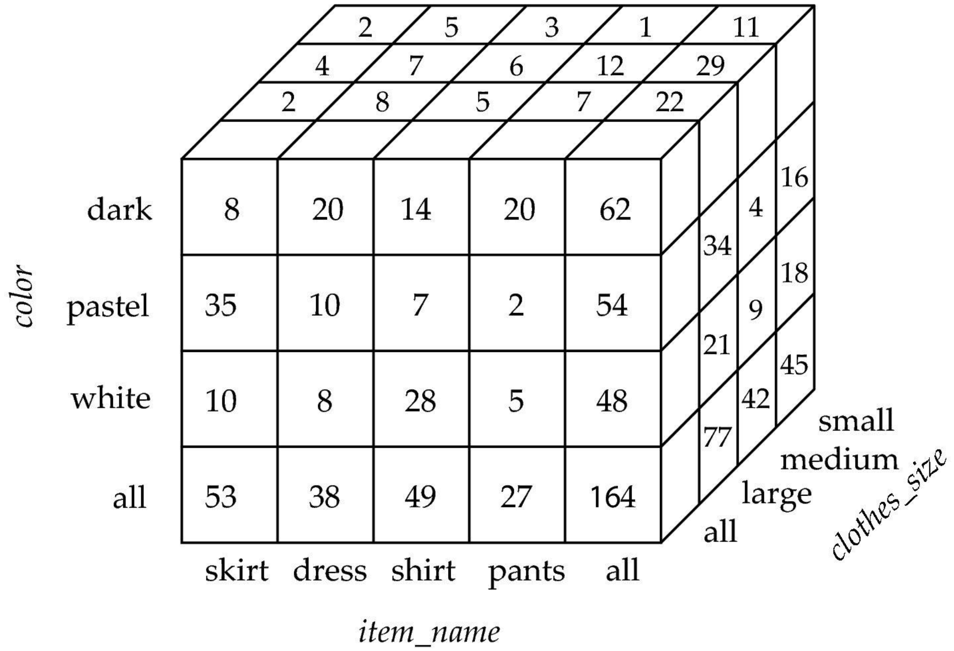
clothes_size **all**

item_name \ *color*	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

- The table above is an example of a **cross-tabulation** (**cross-tab**), also referred to as a **pivot-table**.
  - Values for one of the dimension attributes form the row headers
  - Values for another dimension attribute form the column headers
  - Other dimension attributes are listed on top
  - Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

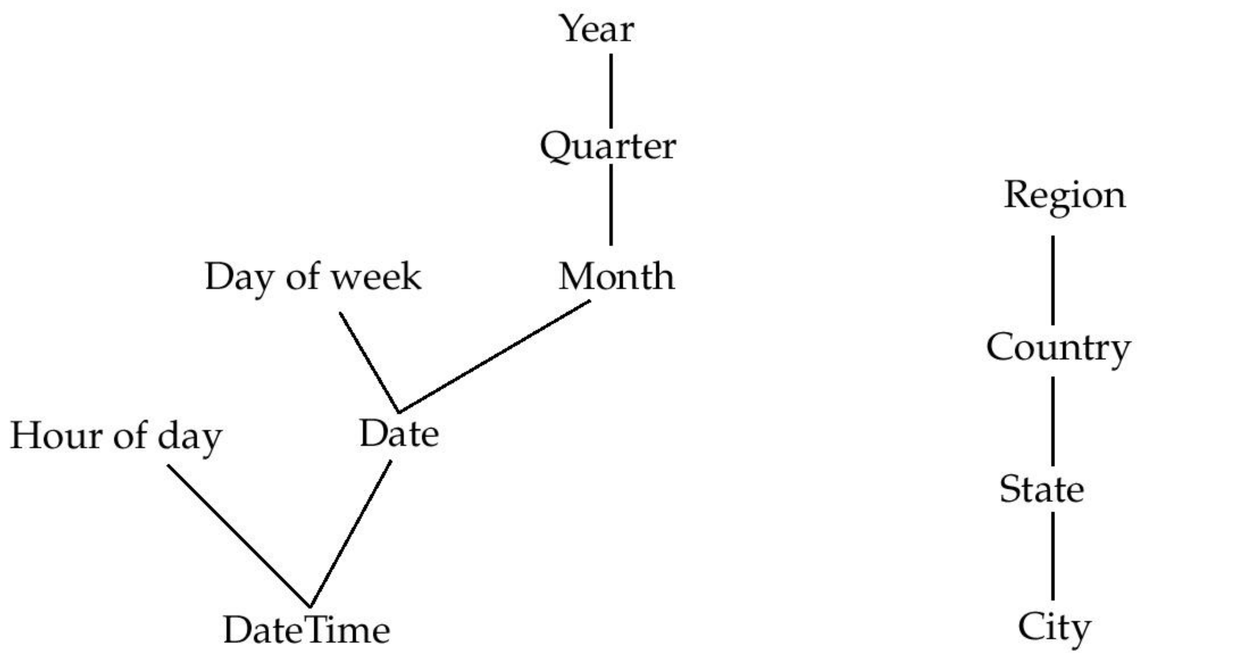
Data Cube

- A **data cube** is a multidimensional generalization of a cross-tab
- Can have *n* dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube



Hierarchies on Dimensions

- **Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail
  - E.g., the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year



a) Time Hierarchy

b) Location Hierarchy

Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
  - Can drill down or roll up on a hierarchy

clothes_size: **all**

category	*item_name*	*color*				
		dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
  - We use the value **all** is used to represent aggregates.
  - The SQL standard actually uses null values in place of **all** despite confusion with regular null values.

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	**all**	8
skirt	pastel	**all**	35
skirt	white	**all**	10
skirt	**all**	**all**	53
dress	dark	**all**	20
dress	pastel	**all**	10
dress	white	**all**	5
dress	**all**	**all**	35
shirt	dark	**all**	14
shirt	pastel	**all**	7
shirt	White	**all**	28
shirt	**all**	**all**	49
pant	dark	**all**	20
pant	pastel	**all**	2
pant	white	**all**	5
pant	**all**	**all**	27
all	dark	**all**	62
all	pastel	**all**	54
all	white	**all**	48
all	**all**	**all**	164

Extended Aggregation to Support OLAP

- The **cube** operation computes union of **group by**'s on every subset of the specified attributes
- Example relation for this section
  sales(*item_name, color, clothes_size, quantity*)
- E.g. consider the query

  > **select** *item_name, color, size,* **sum**(*number*)
  > **from** *sales*
  > **group by cube**(*item_name, color, size*)

  This computes the union of eight different groupings of the *sales* relation:

  { (*item_name, color, size*), (*item_name, color*),
  (*item_name, size*), (*color, size*),
  (*item_name*), (*color*),
  (*size*), ( ) }

  where ( ) denotes an empty **group by** list.
- For each grouping, the result contains the null value for attributes not present in the grouping.

Online Analytical Processing Operations

- Relational representation of cross-tab that we saw earlier, but with *null* in place of **all**, can be computed by

  select *item_name*, *color*, **sum**(*number*)
  from *sales*
  group by cube(*item_name, color*)

- The function **grouping()** can be applied on an attribute
  - Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

  select *item_name, color, size*, **sum**(*number*),
  grouping(*item_name*) **as** *item_name_flag*,
  grouping(*color*) **as** *color_flag*,
  grouping(*size*) **as** *size_flag*,
  from *sales*
  group by cube(*item_name, color, size*)

Online Analytical Processing Operations

- Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**
  - E.g., replace *item_name* in first query by
    decode(**grouping**(item_*name*), 1, ‘all’, *item_name*)

Extended Aggregation (Cont.)

- The **rollup** construct generates union on every prefix of specified list of attributes
- E.g.,

  select *item_name*, *color*, *size*, **sum**(*number*)
  from *sales*
  group by rollup(*item_name, color, size*)

  Generates union of four groupings:

  { (*item_name, color, size*), (*item_name, color*), (*item_name*), ( ) }
- Rollup can be used to generate aggregates at multiple levels of a hierarchy.
- E.g., suppose table *itemcategory*(*item_name, category*) gives the category of each item. Then

  select *category, item_name*, **sum**(*number*)
  from *sales, itemcategory*
  where *sales.item_name = itemcategory.item_name*
  group by rollup(*category, item_name*)

  would give a hierarchical summary by *item_name* and by *category.*

Extended Aggregation (Cont.)

- Multiple rollups and cubes can be used in a single group by clause
  - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,

  select *item_name, color, size*, **sum**(*number*)
  from *sales*
  group by rollup(*item_name*), **rollup**(*color, size*)

  generates the groupings

  {item_name, ()} X {(color, size), (color), ()}

  = { (*item_name, color, size*), (*item_name, color*), (*item_name*), (*color, size*), (*color*), ( ) }

Online Analytical Processing Operations

- **Pivoting:** changing the dimensions used in a cross-tab is called
- **Slicing:** creating a cross-tab for fixed values only
  - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup:** moving from finer-granularity data to a coarser granularity
- **Drill down:** The opposite operation - that of moving from coarser-granularity data to finer-granularity data

OLAP Implementation

- The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems.
- OLAP implementations using only relational database features are called **relational OLAP (ROLAP)** systems
- Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.

OLAP Implementation (Cont.)

- Early OLAP systems precomputed *all* possible aggregates in order to provide online response
  - Space and time requirements for doing so can be very high
    - 2^n combinations of **group by**
  - It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
    - Can compute aggregate on (*item_name, color*) from an aggregate on (*item_name, color, size*)
      - For all but a few “non-decomposable” aggregates such as *median*
      - is cheaper than computing it from scratch
- Several optimizations available for computing multiple aggregates
  - Can compute aggregate on (*item_name, color*) from an aggregate on (*item_name, color, size*)
  - Can compute aggregates on (*item_name, color, size*), (*item_name, color*) and (*item_name*) using a single sorting of the base data

End of Chapter

Figure 5.22

item_name	*clothes_size*	*dark*	*pastel*	*white*
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
pant	small	14	1	3
pant	medium	6	0	0
pant	large	0	1	2

Figure 5.23

item_name	*quantity*
skirt	53
dress	35
shirt	49
pant	27

Figure 5.24

item_name	*color*	*quantity*
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pant	dark	20
pant	pastel	2
pant	white	5

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

Merge statement (now in Chapter 24)

- Merge construct allows batch processing of updates.
- Example: relation *funds_received* (*account_number, amount*) has batch of deposits to be added to the proper account in the *account* relation

```
merge into account as A
      using (select *
             from funds_received as F )
      on (A.account_number = F.account_number )
      when matched then
          update set balance = balance + F.amount
```

End of Chapter 5