



Chapter 5: Advanced SQL

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Database System Concepts

- Chapter 1: Introduction
- **Part 1: Relational databases**
 - Chapter 2: Introduction to the Relational Model
 - Chapter 3: Introduction to SQL
 - Chapter 4: Intermediate SQL
 - [Chapter 5: Advanced SQL](#)
 - Chapter 6: Formal Relational Query Languages
- **Part 2: Database Design**
 - Chapter 7: Database Design: The E-R Approach
 - Chapter 8: Relational Database Design
 - Chapter 9: Application Design
- **Part 3: Data storage and querying**
 - Chapter 10: Storage and File Structure
 - Chapter 11: Indexing and Hashing
 - Chapter 12: Query Processing
 - Chapter 13: Query Optimization
- **Part 4: Transaction management**
 - Chapter 14: Transactions
 - Chapter 15: Concurrency control
 - Chapter 16: Recovery System
- **Part 5: System Architecture**
 - Chapter 17: Database System Architectures
 - Chapter 18: Parallel Databases
 - Chapter 19: Distributed Databases
- **Part 6: Data Warehousing, Mining, and IR**
 - Chapter 20: Data Mining
 - Chapter 21: Information Retrieval
- **Part 7: Specialty Databases**
 - Chapter 22: Object-Based Databases
 - Chapter 23: XML
- **Part 8: Advanced Topics**
 - Chapter 24: Advanced Application Development
 - Chapter 25: Advanced Data Types
 - Chapter 26: Advanced Transaction Processing
- **Part 9: Case studies**
 - Chapter 27: PostgreSQL
 - Chapter 28: Oracle
 - Chapter 29: IBM DB2 Universal Database
 - Chapter 30: Microsoft SQL Server
- **Online Appendices**
 - Appendix A: Detailed University Schema
 - Appendix B: Advanced Relational Database Model
 - Appendix C: Other Relational Query Languages
 - Appendix D: Network Model
 - Appendix E: Hierarchical Model



Chapter 5: Advanced SQL

- 5.1 Accessing SQL From a Programming Language
 - Dynamic SQL
 - ▶ JDBC and ODBC
 - Embedded SQL
- 5.2 Functions and Procedures
- 5.3 Triggers
- 5.4 Recursive Queries**
- 5.5 Advanced Aggregation Features**
- 5.6 OLAP**



JDBC and ODBC

(Dynamic Binding between DB and PL)

- API (application-program interface) for a program to interact with a database server
- Application programs makes API calls to
 - Connect with the database server
 - Send SQL commands to the database server
 - Fetch tuples of result one-by-one into program variables
- Run-Time Binding between DB and PL
- ODBC (Open Database Connectivity)
 - ODBC 1.0 (1992), ODBC 3.0 (1995), ODBC 3.8 (2009)
 - Originally designed for C, now works with C++, C#, and Visual Basic
 - Other API's such as ADO.NET sit on top of ODBC
 - ODBC drivers exist for most DBMSs
- JDBC (Java Database Connectivity) works with Java
 - JDBC driver was a part of Java Development Kit (JDK) 1.1 (1997)
 - Since then, a part of Java Platform, Standard Edition (Java SE)
 - JDBC 4.2 (2013)



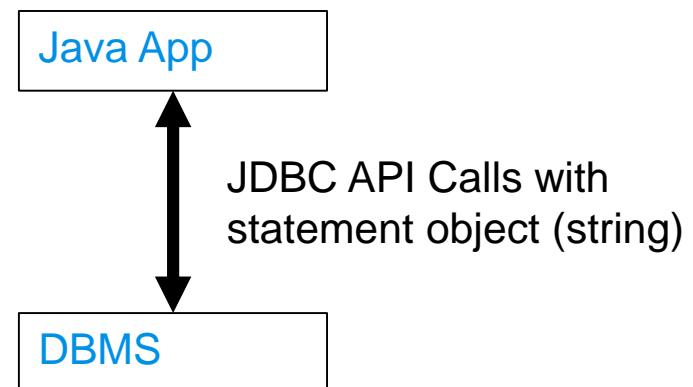
JDBC

■ JDBC is a Java API for communicating with database systems supporting SQL

- supports a variety of features for querying and updating data, and for retrieving query results
- supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Dynamic (Run-time) Binding between application programs and DBMS

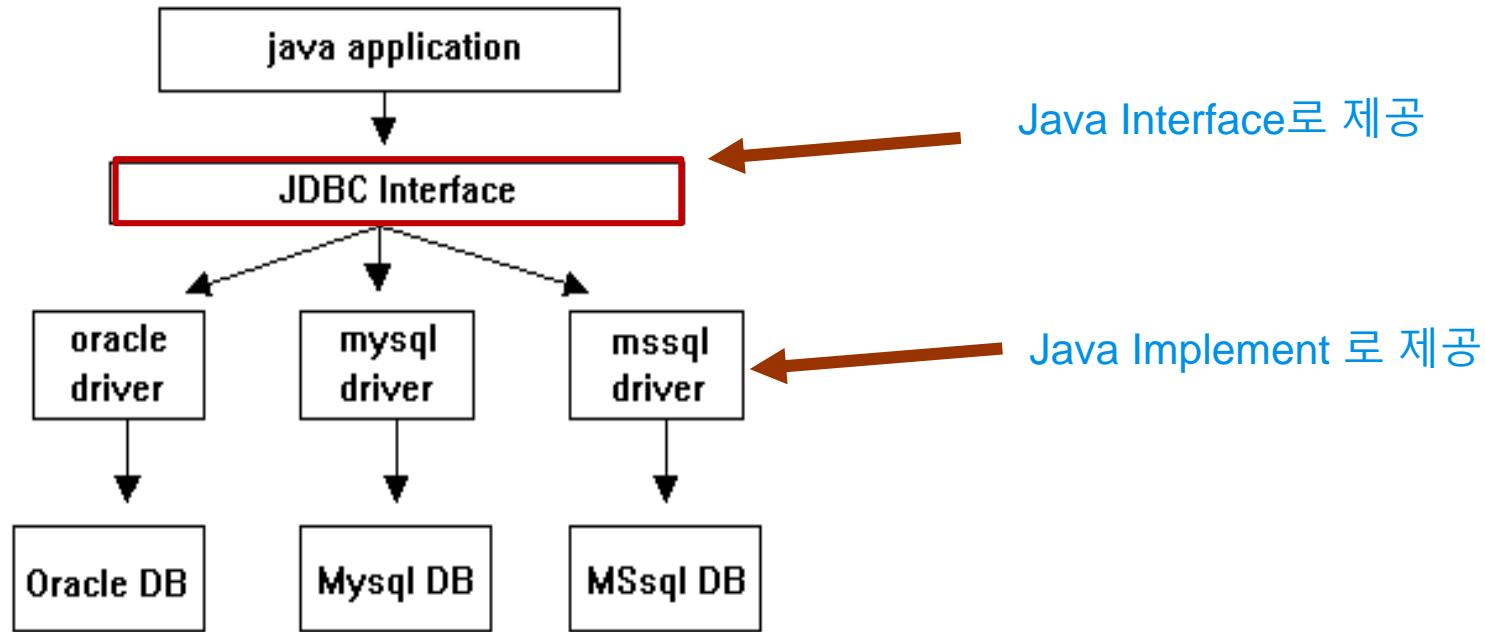
■ JDBC Model for communicating with the database:

- Open a connection
- Create a “statement” object (string)
- Execute queries using the Statement object to send queries and fetch results
- Exception mechanism to handle errors



JDBC란 Java DataBase Connectivity의 약자로 데이터베이스에 연결 및 작업을 하기 위한 자바 표준 인터페이스이다.
자바는 DBMS의 종류에 상관없이 하나의 JDBC API를 사용해서 데이터베이스 작업을 처리할 수 있다.

1. JDBC의 구조



자바언어에서 데이터베이스 표준 인터페이스를 정의하고, 각 데이터베이스 회사들은 JDBC 인터페이스를 제공받아 자신들의 데이터베이스에 맞게 구현한후 드라이버를 제공한다. 개발자가 JDBC API를 사용할 경우 DBMS에 알맞은 JDBC드라이버만 있으면 어떤 데이터베이스라도 사용할 수 있게 된다.

JDBC 드라이버는 각 밴드사에서 jar파일로 제공된다.

```

1. String driver="oracle.jdbc.driver.OracleDriver";
2. String url="jdbc:oracle:thin:@localhost:1521:ORCL";
3. String sql = "select no, name from member";
4. Connection conn = null;
5. PreparedStatement pstmt = null;
6. ResultSet rs = null;
7. try {
8.     // ① JDBC 드라이버 로딩
9.     Class.forName(driver);
10.    // ② 데이터베이스 connection
11.    conn=DriverManager.getConnection(url,"scott","tiger");
12.    // ③ 쿼리(sql)문장을 실행하기 위한 객체 생성
13.    pstmt=conn.prepareStatement(sql);
14.    // ④ 쿼리 실행
15.    rs=pstmt.executeQuery();
16.    // ⑤ 쿼리 실행의 결과값(int, ResultSet) 사용
17.    while(rs.next()){
18.        out.println("<h3>" +rs.getInt(1)+" "+rs.getString(2)+"</h3>");
19.    }
20. }catch(Exception e){
21.     out.println("<h3>데이터 가져오기에 실패하였습니다.</h3>");
22.     e.printStackTrace();
23. }finally{
24.     // ⑥ 사용된 객체 종료
25.     try{rs.close();}catch(Exception e){}
26.     try{pstmt.close();}catch(Exception e){}
27.     try{conn.close();}catch(Exception e){}
28. }

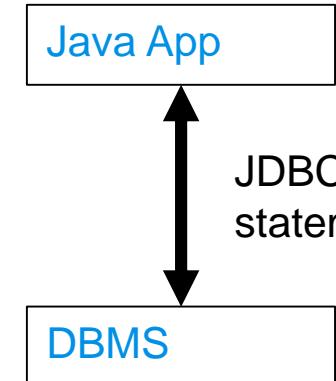
```

Sample Java Code with JDBC Calls

Try-Catch-Finally:

Run-time Error에 대한 Care!

- 어느 상황에서 Error가 발생했는지 detect
- Error가 발생해도 client server connection 유지





JDBC Code

```
public static void JDBCexample(String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver"); // JDBC driver loading
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Update code in the next page....
        ... Query and fetch code in the next page.....
        stmt.close();
        conn.close();
    }
    catch (Exception e) {
        System.out.println("Exception : " + e);
    }
}
```



“Update DB” Code Details

- Update to database

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



“Query & Fetch” Code Details

- Execute query and fetch & print result

```
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) );  
}
```

- Getting result fields:

- **rset.getString(“dept_name”) and rset.getString(1) equivalent if dept_name is the first argument of select result.**



JDBC Feature: Prepared Statement

- Primary Functionality: Activating SQL Precompilation in Serverside DBMS from Clientside
 - DBMS built_in SQL Precompilation
 - ▶ The query is compiled once and then run multiple times with different parameter values
 - PreparedStatement의 ?자리에 나중에 입력되는 value가 들어가는 mechanism

```
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();
```

- Secondary Functionality: quotation mark handling
 - Many PLs and PL environments use single quote and double quote in a mixed manner
 - SQL의 string은 single quote로 둘러 싸여져 있어야 함
 - JDBC Prepared Statement has a special mechanism for handling single quote and double quote in making SQL statement string
 - ▶ Insert \ for a single quotation mark



“Potentially Dangerous” SQL Injection in JDBC [1]

- Java program이 입력창에서 data를 받아서 variable들에 binding하고 DBMS에 아래와 같은 string을 pass했다면
 - “ insert into instructor

```
values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + " )"
```
 - What if name is D'Souza? → single quotation mark will create a syntax error
- Java program이 입력창에서 name를 받아서 name variable에 binding하고 DBMS에 아래와 같은 string을 pass했다면 (SQL에서는 string은 single quote로 둘러싸여져 있어야 함)
 - “ select * from instructor where name = '" + name + "' " ”
- Then, suppose the user, instead of entering a proper name, enters: X' or 'Y' = 'Y
 - then the resulting statement becomes:
 - ▶ “ select * from instructor where name = ' " + " X' or 'Y' = 'Y " + " " ”
 - Then string concatenation이 정리된후에, the query to be inserted into DBMS would be:
 - ▶ Select * from instructor where name = 'X' or 'Y' = 'Y'
 - ▶ DBMS will return everything in instructor
- JDBC Prepared statement internally inserted \ for a single quotation mark:
 - 입력창에 들어온 Data에서 single quote는 \로 교체
 - then the resulting statement becomes:
 - ▶ select * from instructor where name = 'X\' or \'Y\' = \'Y'
 - This query will return an empty relation



“Potentially Dangerous” SQL Injection in JDBC [2]

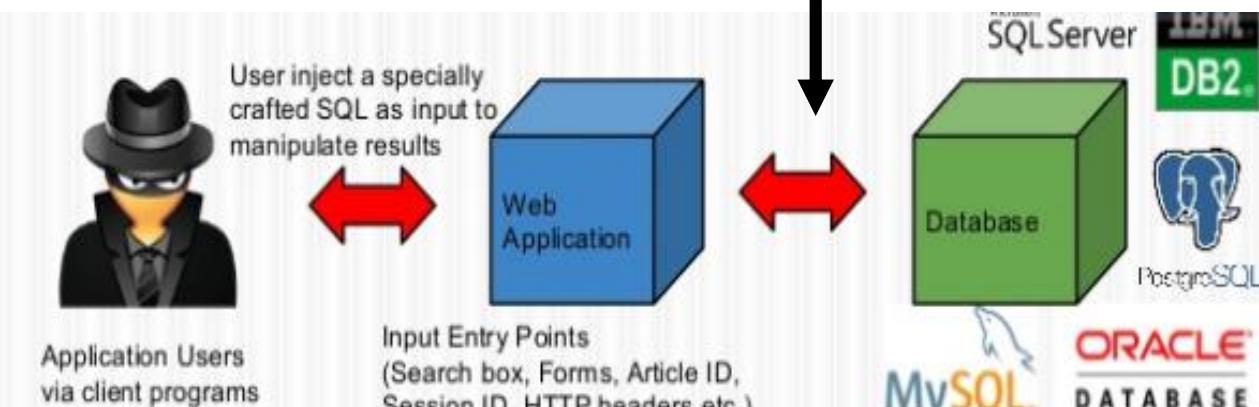
- Even worse scenario!
- Suppose query is constructed using Java
 - “ select * from instructor where name = ” + name + ” ”
- User could have even used instead of entering a name
 - X’ ; update instructor set salary = salary + 10000; ’
 - Then the query to be inserted into DBMS would be:
 - ▶ select * from instructor where name = ‘ X’ ;
 - update instructor set salary = salary + 10000; ’ ’
 - DBMS will increase the salary of every instructor
- JDBC Prepared statement internally uses \ for a single quotation mark:
 - ▶ select * from instructor where name = ‘ X\’ ;
 - update instructor set salary = salary + 10000; \’
 - This query will return an empty relation
- JDBC 의 PreparedStatement 기능은 Malicious user가 single quote를 가지고 위험한 string을 만들어도 execution이 안되도록 control 한다



SQL Injection

JDBC API Calls with statement object (**string**)

- SQL 문장은 ; 로 나누어짐
- SQL에서 string은 single quote로 표현됨



Username	Password	Permit
Kim	5252	OK
Bob	1234	OK
John	2888	OK

Enter your username and password below, then press Login.

Username:

Password:

Login

“ Select Permit
From User-Check
Where Username = ‘Bob’
and Password = ‘1234’
”



JDBC Feature: Meta Data Handling [1/2]

- Two types of JDBC Metadata

- Meta data for a query
 - Meta data for the whole database

- ResultSetMetadata: meta data for query result

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? ➔ useful for printing out the formatted table for query result!



JDBC Feature: Meta Data Handling [2/2]

- DatabaseMetadata: meta data for the database itself

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern, 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? → can be used for a database browser (or database GUI) that allows a user to navigate or examine tables



JDBC Feature: Transaction Control

■ Updatable result sets

- 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);`
- A group of transactions must then be committed or rolled back explicitly
 - ▶ `conn.commit();` or
 - ▶ `conn.rollback();`
- `conn.setAutoCommit(true)` turns on automatic commit
- Clientside JAVA application에서 Serverside DBMS에 있는 transaction control 관련 command를 activate 시킬수 있다



Additional JDBC Features

- **CallableStatement: invoking SQL stored functions and procedures**
 - `CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");`
 - `CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)?}");`
 - Java program 내부에서 cStmt1와 cStmt2를 수행할수 있다

- **Handling large object types without creating the large object in memory**
 - `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).`
 - Database에 있는 large object들을 Java program에서 partial access, update등이 가능하게 한다



ODBC [1/2]

- Open Data Base Connectivity(ODBC) standard
 - Initially **C language connection API** to DBMS
 - 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



ODBC [2/2]

- Each database system supporting ODBC provides a “[ODBC 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\(\)](#) with parameters as below
 - 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

[1/3]

```
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);
    { ...Code in the next page... }

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



ODBC Code

[2/3]

■ 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 Code

[3/3]

- Program sends SQL commands to the 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 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.

■ Transaction control: 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)`
- A group of transactions can 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

- MicroSoft Database connection API for Visual Basic .NET and C#

(part of MS .NET framework, 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 first 2 attributes of result*/
}
rdr.Close();
conn.Close();
```

- Translated into ODBC calls

- Can also access non-relational data sources such as

- OLE-DB
- XML data
- Entity framework



Embedded SQL [1/2]

(Static Binding of DB and PL)

- The SQL standard defines embeddings of SQL in PLs such as C, Java, and Cobol
 - ESQL/C, ESQL/Java, ESQL/Cobol
 - Compile-time binding between application program and relational DBMS
- 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*
- System R: ESQL into PL/I, Most DBMSs: **ESQL/C**
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor
EXEC SQL <embedded SQL statement> ;
- From within a host language (Eg, C), find the ID and name of students who have completed more than the number of credits stored in variable **credit_amount**

Step1: 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 ;
```



Embedded SQL

[2/2]

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

EXEC SQL open c ;

- 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

- 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 ;

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



Updates Through Cursors in Embedded SQL

- 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 ;
```

- To update tuple at the current location of cursor c

EXEC SQL

```
update instructor
set salary = salary + 100
where current of c ;
```



SQLJ [1/4]

- Alternative approach for ESQL/Java
- JDBC에 비해서 Java Syntax에 가깝게 표현되는 방식
- JDBC에 대한 wrapper로 보는것이 적절

- SQLJ part 0: ANSI (1998), "Object Language Bindings (SQL/OLB)", SQL—Part 10
- SQLJ part 1: ANSI (1999), "SQL Routines Using the Java"
- SQLJ part 2: ANSI (2000), "SQL Types Using the Java"

- *Object Language Bindings (SQL/OLB)* : ISO (2000), SQL—Part 10
- *SQL Routines and Types Using the Java (SQL/JRT)*: ISO (2002), SQL—Part 13

- *Only DBMSs by IBM, Oracle support SQLJ*
 - *Translator from SQLJ code to JCBC code*



SQLJ

[2/4]

- JDBC is overly dynamic, errors cannot be caught by compiler
 - 그래서 JDBC db application program들은 Try ... Catch..로 구성된다
- ESQL/C와 다른점: EXEC SQL 대신에 #sql, cursor 대신에 iter
- 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();
```



# SQLJ [3/4]

- Some advantages of SQLJ over JDBC include:
  - SQLJ commands tend to be **shorter** than equivalent JDBC programs
  - SQL syntax can be checked at **compile time**
  - The returned query results can also be checked strictly
  - Preprocessor might generate static SQL which performs better than dynamic SQL because query plan is created on program compile time, stored in database and reused at runtime
  
- Disadvantages include:
  - SQLJ requires **a preprocessing step**
  - Many IDEs **do not have SQLJ support**



# SQLJ

[4/4]

## Single-row query

### JDBC

```
PreparedStatement stmt = conn.prepareStatement(
 "SELECT MAX(SALARY), AVG(SALARY)"
 + " FROM DSN8710.EMP");
rs = stmt.executeQuery();
if (!rs.next()) {
 // Error--no rows found
}
maxSalary = rs.getBigDecimal(1);
avgSalary = rs.getBigDecimal(2);
if (rs.next()) {
 // Error--more than one row found
}
rs.close();
stmt.close();
```

### SQLJ

```
#sql [ctx] {
 SELECT MAX(SALARY), AVG(SALARY)
 INTO :maxSalary, :avgSalary
 FROM DSN8710.EMP
};
```



# Chapter 5: Advanced SQL

- 5.1 Accessing SQL From a Programming Language
  - Dynamic SQL
    - ▶ JDBC and ODBC
  - Embedded SQL
- 5.2 Functions and Procedures
- 5.3 Triggers
- 5.4 Recursive Queries\*\*
- 5.5 Advanced Aggregation Features\*\*
- 5.6 OLAP\*\*



# 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**
- OOP의 method 개념을 support
- Object-oriented aspects of these features are covered in Chapter 22 (Object Based Databases)

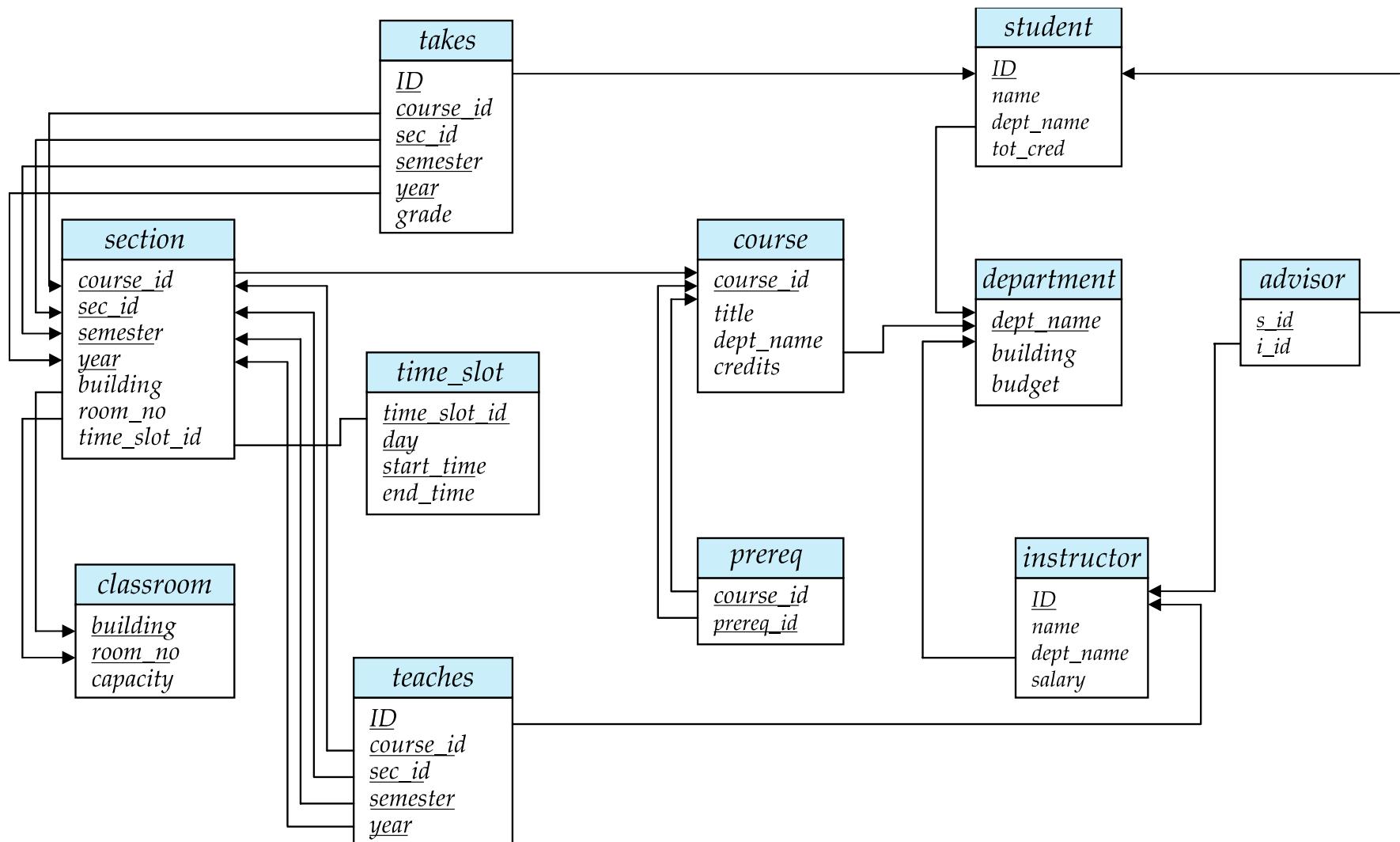


# Functions and Procedures

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external PL
  - 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 DBMSs have **proprietary procedural extensions to SQL** that differ from SQL:1999.



# Schema Diagram for University Database





# SQL Functions

- Input parameter declaration, Output data type declaration
- 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
```

- Calling a function within a query: Find the department name and budget of all departments with more than 12 instructors.

```
select dept_name, budget
from department
where dept_count (dept_name) > 12
```



# SQL Table Functions

- SQL:2003 added functions that return a relation as a result
- Define a **table function**: 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)
```

- Calling a **table function**:

```
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



# SQL Procedural Constructs [1/3]

- Warning: most DBMSs 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
```

## ■ For loop

Permits iteration over all results of a query

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



# SQL Procedural Constructs [2/3]

- SQL:1999 Conditional statements: **if-then-else** and **case**
- Example procedure: registers student after ensuring classroom capacity is not exceeded:  
Returns 0 on success and -1 if capacity is exceeded

```
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;
```



# SQL Procedural Constructs [3/3]

- Function이나 Procedure의 내부에 Exception Handling mechanism을 심을수 있다
  - SQL Shell에서 run-time error에 대한 대비책
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
...
some code
.....
if (자리가 더 이상 없으면) then signal out_of_classroom_seats
else continue
end
```

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



# Functions/Procedures written in External PL [1/2]

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

```
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'
```

여기부터는 SQL밖에서 Function과 Procedure를 만들어서  
SQL안에서 call하는것을 allow하는 방식!



# Functions/Procedures written in External PL [2/2]

- 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 allowed when efficiency is more important than security.

Convenience + Performance vs. Security



# Security with External Language Routines

- To deal with security problems of SQL Functions and Proceure
  - A safe language like **Java** has the **sandbox** feature which cannot be used to access/damage other parts of the database code.
  - **Some languages** run external language functions/procedures **in a separate process (different address space)**, 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.
- **Java Sand Box Model:** Java Compiler가 신뢰성이 확보되지 않은 코드를 load할 때 항상 바이트 코드를 검사하여, 신용이 확인되지 않은 코드는 자바환경전체에 아무런 해약이 미치지 못하는 'Sand Box'에서 수행시켜 Local System에 대한 전면적인 접근을 금지하는 기법.
- SQL address space 와 외부PL function의 address space를 분리



# Java Security Model: Sandbox [1/2]

- 외부에서 받은 프로그램을 SandBox라는 보호된 영역을 가진 프로그램을 만들어 프로그램설치를 SandBox 안에 가둔 뒤 작동시키는 방법으로 프로그램의 폭주 또는 악성 바이러스의 침투를 막는다

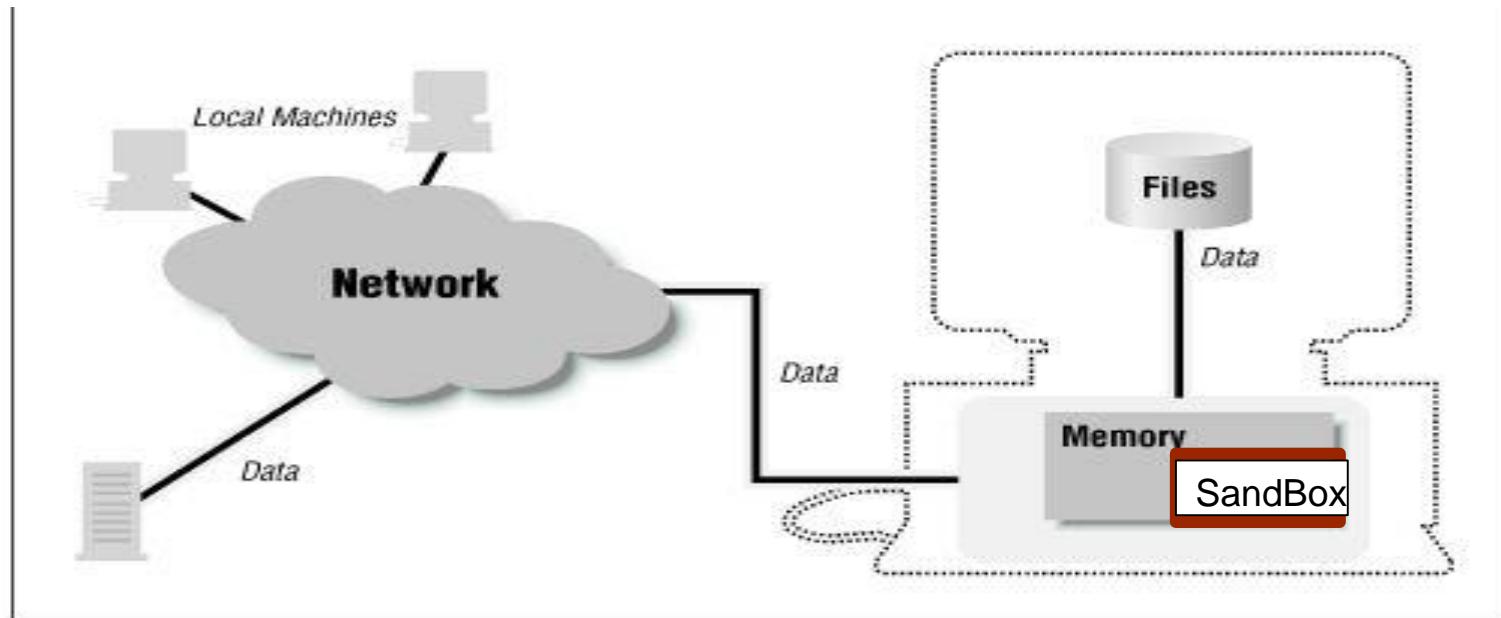


Figure 1-1. A machine has access to many resources



# Java Security Model: Sandbox [2/2]

- Different-sized sandboxes in which a Java program or Applet might run:
- A sandbox in which the program has access to the CPU, the screen, keyboard, and mouse, and to its own memory.
  - This is the minimal sandbox--it contains just enough resources for a program to run.
- A sandbox in which the program has access to the CPU and its own memory as well as access to the web server from which it was loaded.
  - This is often thought of as the default state for the sandbox.
- A sandbox in which the program has access to the CPU, its memory, its web server, and to a set of program-specific resources (local files, local machines, etc.).
  - A word-processing program, for example, might have access to the *docs* directory on the local filesystem, but not to any other files.
- An open sandbox, in which the program has access to whatever resources the host machine normally has access to.



# Chapter 5: Advanced SQL

- 5.1 Accessing SQL From a Programming Language
  - Dynamic SQL
    - ▶ JDBC and ODBC
  - Embedded SQL
- 5.2 Functions and Procedures
- 5.3 Triggers
- 5.4 Recursive Queries\*\*
- 5.5 Advanced Aggregation Features\*\*
- 5.6 OLAP\*\*



# 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 were officially introduced to SQL standard in **SQL:1999**
- But triggers have been 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 [1/2]

- 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;**

**section**

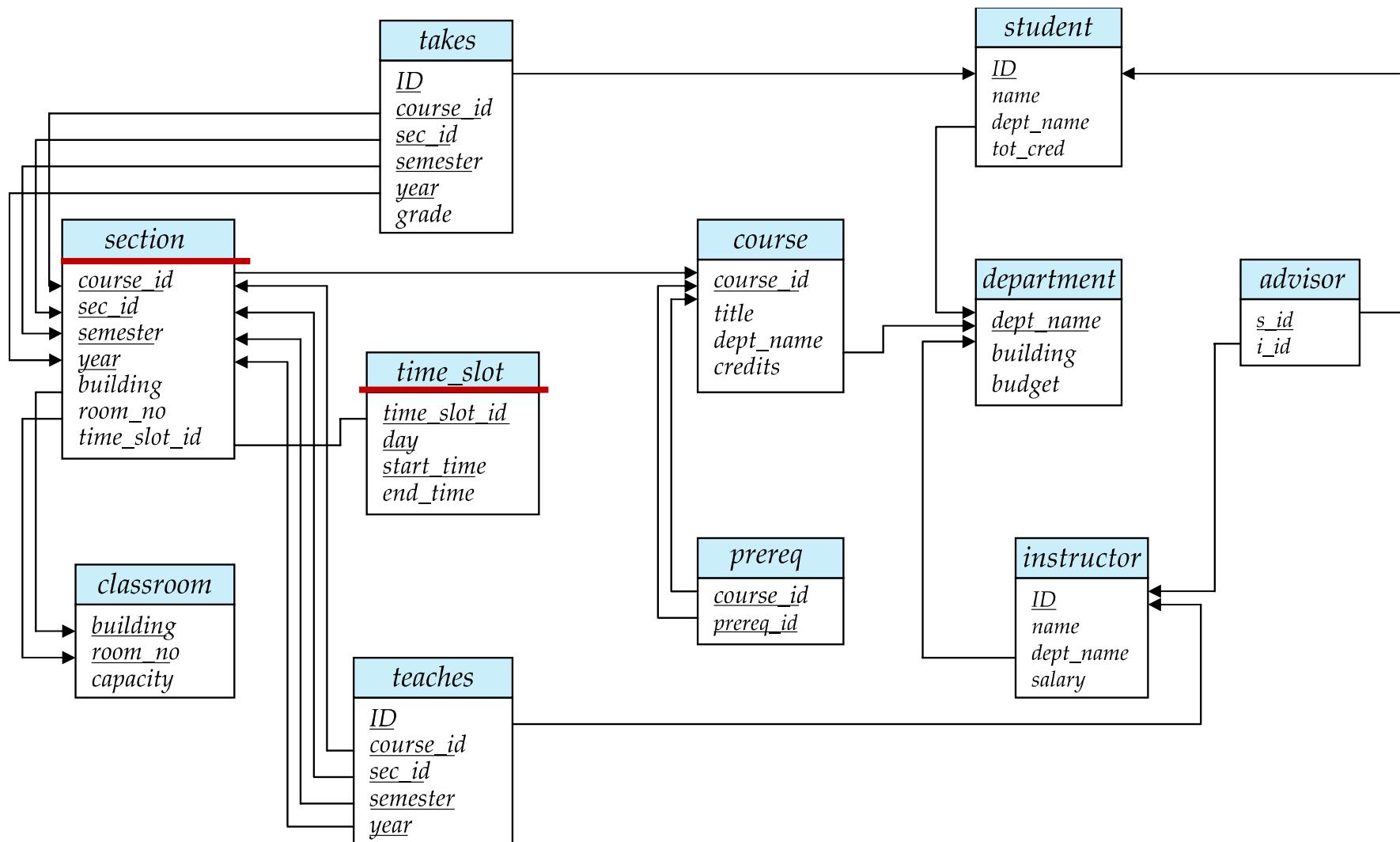
| course_id | sec_id | semester | year | building | room_number | time_slot_id |
|-----------|--------|----------|------|----------|-------------|--------------|
| BIO-101   | 1      | Summer   | 2009 | Painter  | 514         | B            |
| BIO-301   | 1      | Summer   | 2010 | Painter  | 514         | A            |
| CS-101    | 1      | Fall     | 2009 | Packard  | 101         | H            |
| CS-101    | 1      | Spring   | 2010 | Packard  | 101         | F            |
| CS-190    | 1      | Spring   | 2009 | Taylor   | 3128        | E            |

**Time\_slot**

| time_slot_id | day | start_time | end_time |
|--------------|-----|------------|----------|
| A            | M   | 8:00       | 8:50     |
| A            | W   | 8:00       | 8:50     |
| A            | F   | 8:00       | 8:50     |
| B            | M   | 9:00       | 9:50     |
| B            | W   | 9:00       | 9:50     |



# Schema Diagram for University Database





# Trigger Example [2/2]

```
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;
```

**section**

| <i>course_id</i> | <i>sec_id</i> | <i>semester</i> | <i>year</i> | <i>building</i> | <i>room_number</i> | <i>time_slot_id</i> |
|------------------|---------------|-----------------|-------------|-----------------|--------------------|---------------------|
| BIO-101          | 1             | Summer          | 2009        | Painter         | 514                | B                   |
| BIO-301          | 1             | Summer          | 2010        | Painter         | 514                | A                   |
| CS-101           | 1             | Fall            | 2009        | Packard         | 101                | H                   |
| CS-101           | 1             | Spring          | 2010        | Packard         | 101                | F                   |
| CS-190           | 1             | Spring          | 2009        | Taylor          | 3128               | E                   |

**Time\_slot**

| <i>time_slot_id</i> | <i>day</i> | <i>start_time</i> | <i>end_time</i> |
|---------------------|------------|-------------------|-----------------|
| A                   | M          | 8:00              | 8:50            |
| A                   | W          | 8:00              | 8:50            |
| A                   | F          | 8:00              | 8:50            |
| B                   | M          | 9:00              | 9:50            |
| B                   | W          | 9:00              | 9:50            |



# 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 can be referenced **before and after an update**
  - **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;
```

| takes |           |        |          |      |       |
|-------|-----------|--------|----------|------|-------|
| ID    | course_id | sec_id | semester | year | grade |
| 00128 | CS-101    | 1      | Fall     | 2009 | A     |
| 00128 | CS-347    | 1      | Fall     | 2009 | A-    |
| 12345 | CS-101    | 1      | Fall     | 2009 | C     |
| 12345 | CS-190    | 2      | Spring   | 2009 | A     |
| 12345 | CS-315    | 1      | Spring   | 2010 | A     |
| 12345 | CS-347    | 1      | Fall     | 2009 | A     |



# Trigger to Maintain credits\_earned value

- Keep `tot_cred` up-to-date when grade attribute is updated
- 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;

| student         |                   |                        |                       |
|-----------------|-------------------|------------------------|-----------------------|
| <code>ID</code> | <code>name</code> | <code>dept_name</code> | <code>tot_cred</code> |
| 00128           | Zhang             | Comp. Sci.             | 102                   |
| 12345           | Shankar           | Comp. Sci.             | 32                    |
| 19991           | Brandt            | History                | 80                    |
| 23121           | Chavez            | Finance                | 110                   |

| takes           |                        |                     |                       |                   |                    |
|-----------------|------------------------|---------------------|-----------------------|-------------------|--------------------|
| <code>ID</code> | <code>course_id</code> | <code>sec_id</code> | <code>semester</code> | <code>year</code> | <code>grade</code> |
| 00128           | CS-101                 | 1                   | Fall                  | 2009              | A                  |
| 00128           | CS-347                 | 1                   | Fall                  | 2009              | A-                 |
| 12345           | CS-101                 | 1                   | Fall                  | 2009              | C                  |
| 12345           | CS-190                 | 2                   | Spring                | 2009              | A                  |
| 12345           | CS-315                 | 1                   | Spring                | 2010              | A                  |
| 12345           | CS-347                 | 1                   | Fall                  | 2009              | A                  |



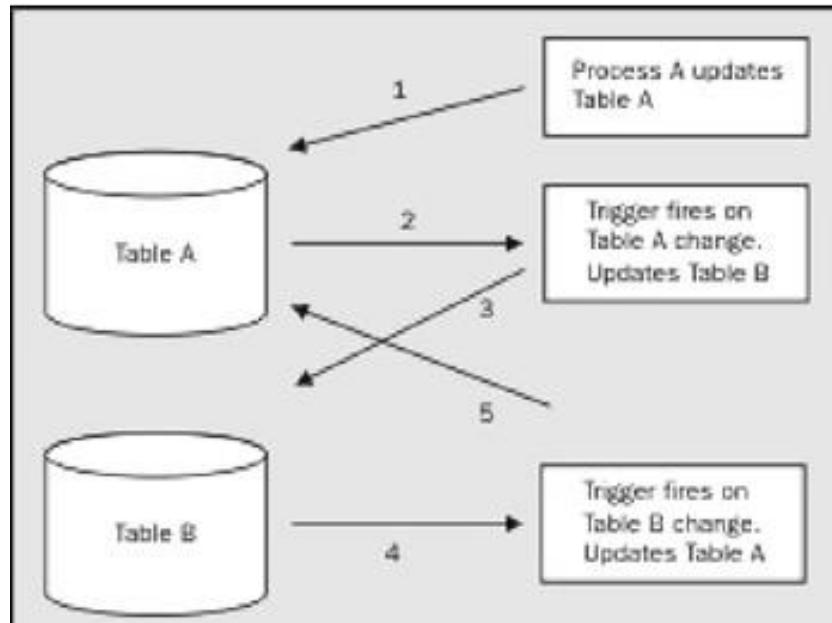
# Statement Level Triggers

- 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



# Alternatives to Triggers [1/2]

- Triggers were used earlier for tasks such as
  - maintaining a derived attribute such as **summary data** (ex. 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
- Triggers should be written **with great care**
  - Loading data from a backup copy and replicating updates at a remote site may cause **unintended execution of triggers**
  - Trigger runtime error leading to failure of critical transactions that set off the trigger
  - Cascading execution of a chain of triggers may cause complication





# Alternatives to Triggers [2/2]

- Triggers are **best avoided** when alternatives exist
- There are **better ways of doing these** in modern-day DBMS:
  - Integrity constraint enforcement를 위한 Trigger
    - ▶ Built-in DBMS **materialized view** facilities to maintain summary data
  - Multiple data copy들의 integrity maintenance를 위한 Trigger
    - ▶ Built-in DBMS support for **data replication**
  - Built-in DBMS support가 없는 상황에서도 Trigger를 사용하기 보다는
    - ▶ **Function or procedures for update** may encapsulate necessary actions
- 왜 그럴까?
  - Trigger는 user가 임의로 구현하므로 복잡하게 구현하다 보면 losing control
  - 가능하면 Built-in DBMS 기능으로 문제를 해결하는 게 바람직



# Chapter 5: Advanced SQL

- 5.1 Accessing SQL From a Programming Language
  - Dynamic SQL
    - ▶ JDBC and ODBC
  - Embedded SQL
- 5.2 Functions and Procedures
- 5.3 Triggers
- 5.4 Recursive Queries\*\*
- 5.5 Advanced Aggregation Features\*\*
- 5.6 OLAP\*\*



# Recursive View in SQL

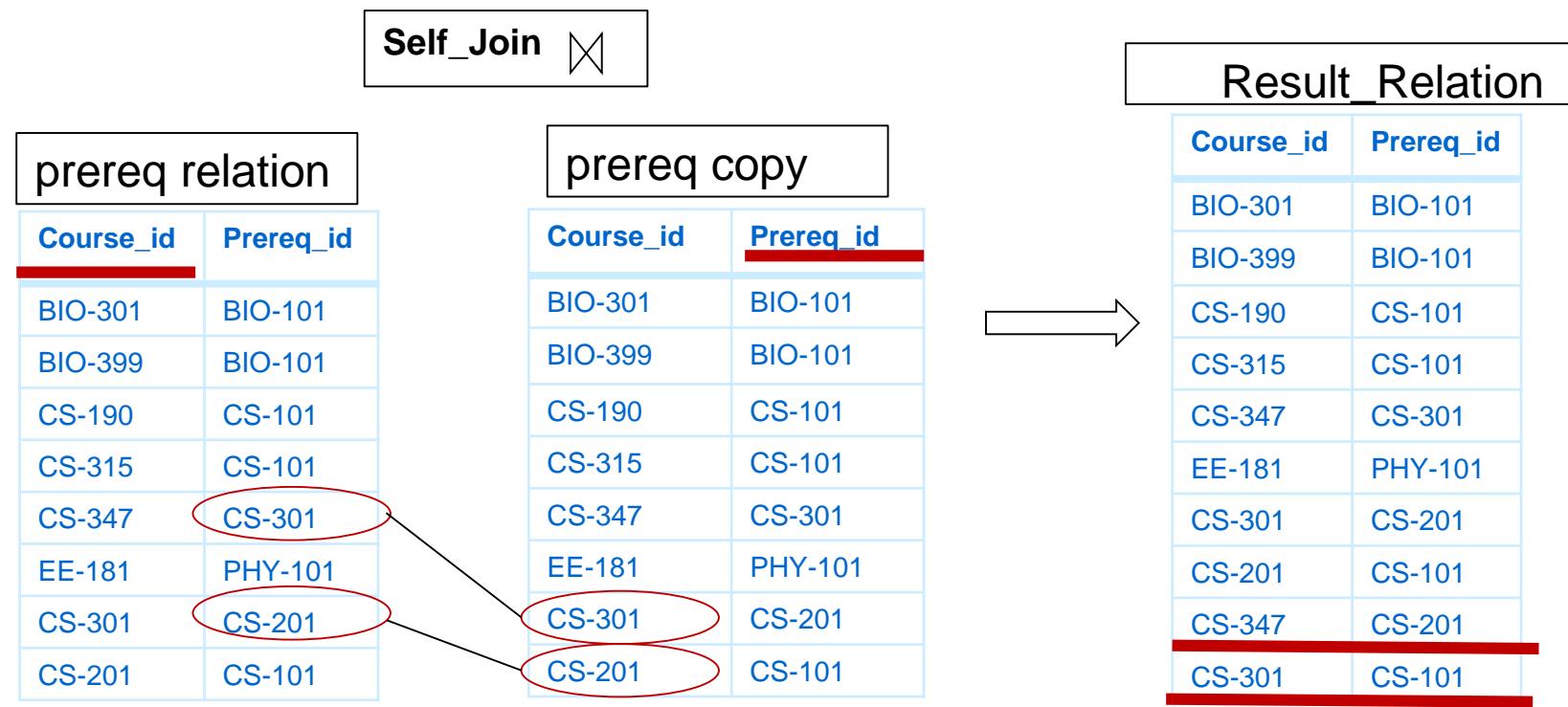
- SQL:1999 permits recursive view definition
- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive c_prereq(course_id, prereq_id) as (
 select course_id, prereq_id
 from prereq
 union
 select prereq.prereq_id, c_prereq.course_id
 from prereq, c_prereq
 where prereq.course_id = c_prereq.prereq_id
)
select *
from c_prereq;
```

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



# Transitive Closure Computation on a Relation [1/3]



2개의 새로운 tuple이 생성



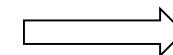
# Transitive Closure Computation on a Relation [2/3]

새로 생성된 Tuple들만  
가지고 Result\_Relation과  $\bowtie$

|        |        |
|--------|--------|
| CS-347 | CS-201 |
| CS-301 | CS-101 |

| Course_id | Prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| BIO-399   | BIO-101   |
| CS-190    | CS-101    |
| CS-315    | CS-101    |
| CS-347    | CS-301    |
| EE-181    | PHY-101   |
| CS-301    | CS-201    |
| CS-201    | CS-101    |
| CS-347    | CS-201    |
| CS-301    | CS-101    |

Result\_Relation



| Course_id | Prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| BIO-399   | BIO-101   |
| CS-190    | CS-101    |
| CS-315    | CS-101    |
| CS-347    | CS-301    |
| EE-181    | PHY-101   |
| CS-301    | CS-201    |
| CS-201    | CS-101    |
| CS-347    | CS-201    |
| CS-301    | CS-101    |
| CS-347    | CS-101    |

1개의 새로운 tuple이 생성



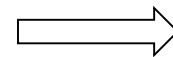
# Transitive Closure Computation on a Relation [3/3]

새로 생성된 Tuple만  
가지고 Result\_Relation과  $\bowtie$

|        |        |
|--------|--------|
| CS-347 | CS-101 |
|--------|--------|

?

| Course_id | Prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| BIO-399   | BIO-101   |
| CS-190    | CS-101    |
| CS-315    | CS-101    |
| CS-347    | CS-301    |
| EE-181    | PHY-101   |
| CS-301    | CS-201    |
| CS-201    | CS-101    |
| CS-347    | CS-201    |
| CS-301    | CS-101    |
| CS-347    | CS-101    |



Result\_Relation

| Course_id | Prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| BIO-399   | BIO-101   |
| CS-190    | CS-101    |
| CS-315    | CS-101    |
| CS-347    | CS-301    |
| EE-181    | PHY-101   |
| CS-301    | CS-201    |
| CS-201    | CS-101    |
| CS-347    | CS-201    |
| CS-301    | CS-101    |
| CS-347    | CS-101    |

새로운 tuple이 생성되지  
않으므로 최종 Result\_Relation



# Non-Recursive Method for Transitive Closure Query

- 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
- Computing transitive closure using iteration, adding successive tuples to *c\_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *c\_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 *c\_prereq* contains all of the tuples it contained before, plus possibly more

```

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

```

Mission: Finding all prerequisites of a course!

- 1999년 이전에는 SQL만 가지고 불가능
- 1999에 SQL1999 (SQL3)에서 Procedural extension과 Recursive View 를 지원하면서 Transitive Closure 가능

Using procedural constructs SQL1999 such as [repeat..until](#)



# Computing transitive closure using iteration on a Relation

prereq relation

| Course_id | Prereq_id |
|-----------|-----------|
| BIO-301   | BIO-101   |
| BIO-399   | BIO-101   |
| CS-190    | CS-101    |
| CS-315    | CS-101    |
| CS-347    | CS-301    |
| EE-181    | PHY-101   |
| CS-301    | CS-201    |
| CS-201    | CS-101    |

Prerequisite courses of cs-347

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

Fixed Point

CS-347, CS-301



CS-347, CS-201



CS-347, 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.employee_name)
select *
from empl
```

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



# Chapter 5: Advanced SQL

- 5.1 Accessing SQL From a Programming Language
  - Dynamic SQL
    - ▶ JDBC and ODBC
  - Embedded SQL
- 5.2 Functions and Procedures
- 5.3 Triggers
- 5.4 Recursive Queries\*\*
- 5.5 Advanced Aggregation Features\*\*
- 5.6 OLAP\*\*



# Ranking [1/3]

- Ranking is done in conjunction with an `order by` specification.
- Suppose we have a relation `student_grades(ID, GPA)`
- 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 can be done using basic SQL aggregation, but resultant query is 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 [2/3]

- Ranking can be done **within partition of the data** with **partition by** construct
- “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.
- 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 [3/3]

- 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;
```
- For a given constant  $n$ , 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 [1/2]

- 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`
- Examples of other window specifications:
  - **between rows unbounded preceding and current**
    - ▶ Between을 써서 지금까지의 모든 row들을 표현
  - **rows unbounded preceding**
    - ▶ 지금까지의 모든 rows
  - **range between 10 preceding and current row**
    - ▶ 현재 값과 현재 값 – 10 사이에 있는 값들을 표현
  - **range interval 10 day preceding**
    - ▶ current row를 안쓰고 지난 10일간을 표현



# Windowing [2/2]

- Can do windowing within partitions with `partition by` construct
- 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
```

→ `(account_number, date_time, balance)` 생성



# Chapter 5: Advanced SQL

- 5.1 Accessing SQL From a Programming Language
  - Dynamic SQL
    - ▶ JDBC and ODBC
  - Embedded SQL
- 5.2 Functions and Procedures
- 5.3 Triggers
- 5.4 Recursive Queries\*\*
- 5.5 Advanced Aggregation Features\*\*
- 5.6 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 **quantity**(*number*) of the *sales* relation

- **Dimension attributes**

- ▶ define the dimensions on which measure attributes (or aggregates thereof) are viewed
  - ▶ e.g., the attributes *item\_name*, *color*, and *size* of the *sales* relation

## ■ OLAP on top of DBMS

## ■ OLAP inside DBMS



# Example: sales relation

Dimension attributes

Measure attribute

| 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        |
| shirt     | dark   | large        | 6        |
| shirt     | pastel | small        | 4        |
| shirt     | pastel | medium       | 1        |
| shirt     | pastel | large        | 2        |
| shirt     | white  | small        | 17       |
| shirt     | white  | medium       | 1        |
| shirt     | white  | large        | 10       |
| pant      | dark   | small        | 14       |
| pant      | dark   | medium       | 6        |
| pant      | dark   | large        | 0        |
| pant      | pastel | small        | 1        |
| pant      | pastel | medium       | 0        |
| pant      | pastel | large        | 1        |
| pant      | white  | small        | 3        |
| pant      | white  | medium       | 0        |
| pant      | white  | large        | 2        |

Item\_name 4가지

Color 3가지

Clothes\_size 3가지

총 36 combinations

→ 36 tuples



# Plain Queries for Basic Summary

| <i>item_name</i> | <i>quantity</i> |
|------------------|-----------------|
| skirt            | 53              |
| dress            | 35              |
| shirt            | 49              |
| pants            | 27              |

Figure 5.23 Query result.

```
select item_name, sum(quantity)
from sales
group by item_name;
```

| <i>item_name</i> | <i>color</i> | <i>quantity</i> |
|------------------|--------------|-----------------|
| 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              |
| pants            | dark         | 20              |
| pants            | pastel       | 2               |
| pants            | white        | 5               |

```
select item_name, color, sum(quantity)
from sales
group by item_name, color;
```



# Cross Tabulation of sales by item\_name and color

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        |

clothes\_size all

color

| item_name | 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   |

```
select item_name, color, sum(quantity)
from sales
group by cube(item_name, color)
```

이런 SQL Query의 도움을 받아서!!!!

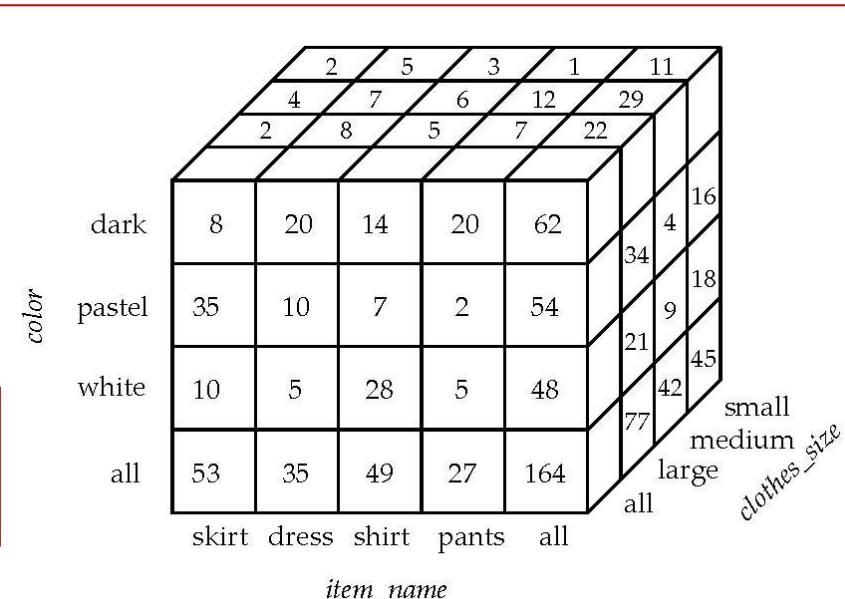
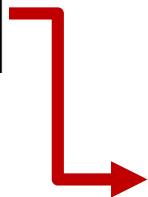
- The table above is an example of a **cross-tabulation (cross-tab)**, also referred to as a **pivot-table**.
- The table is a **very typical summary table or report table** in real world office environments



# RDBMS vs OLAP

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        |



3D Data Cube of sales relation



# OLAP Operations

- **Cube:** union of groups of **every subset combination** of specified attributes
- **Pivoting:** changing the dimensions used in a cross-tab
- **Slicing:** creating a cross-tab for fixed values only
  - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup:**
  - **union of groups of every prefix combination** of specified attributes
  - 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



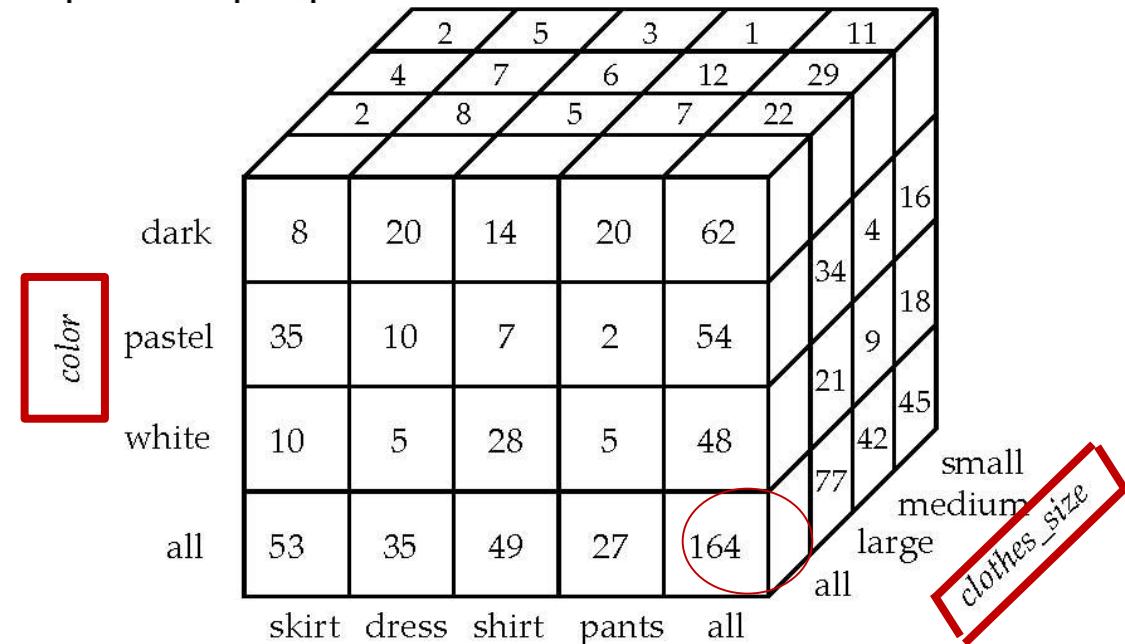
# Data Cube

- 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

| item_name | color | clothes_size | quantity |
|-----------|-------|--------------|----------|
| skirt     | dark  | small        | 2        |
| skirt     | dark  | medium       | 5        |
| skirt     | dark  | large        | 1        |

4 items, 3 colors, 3 sizes  
need  $(4+1) \times (3+1) \times (3+1) = 80$  cells in the data cube

Dimension attributes  
Measure attribute



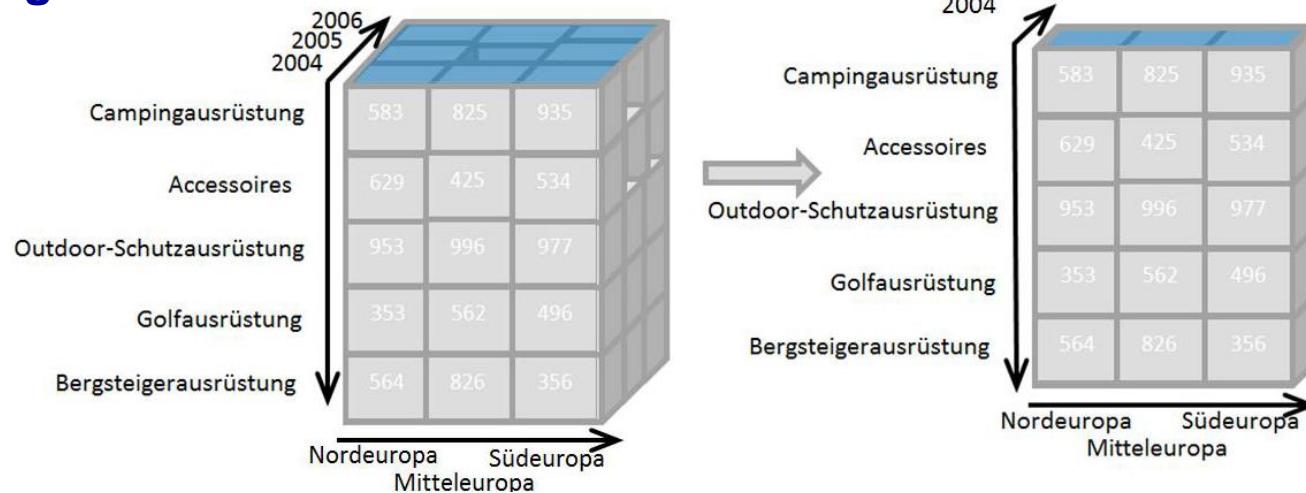
```
select item_name, color, clothes_size, sum(quantity)
from sales
group by cube(item_name, color, clothes_size);
```

item\_name

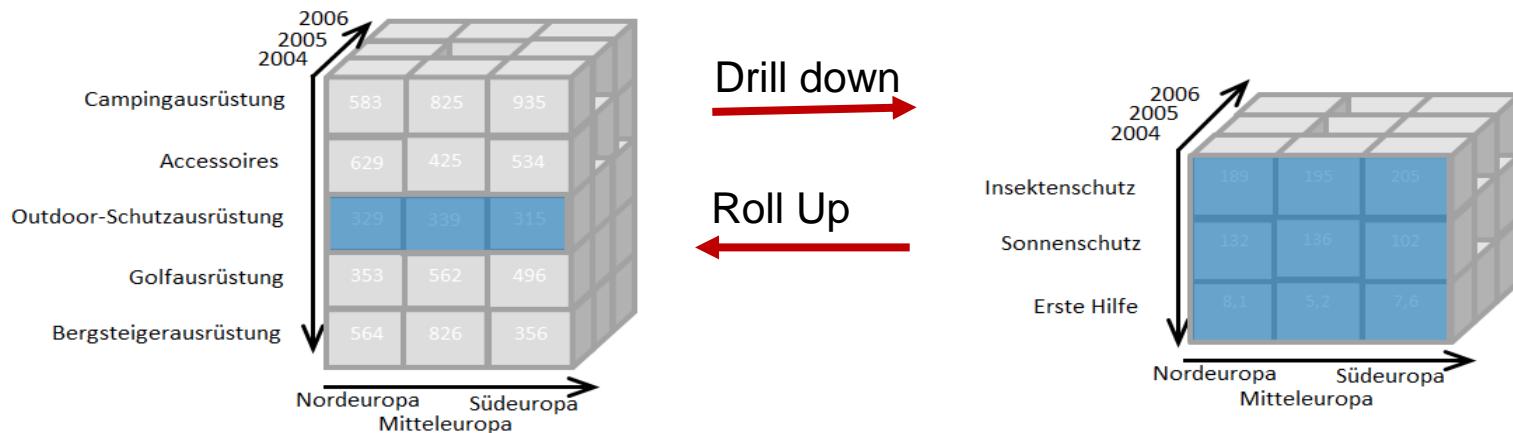


# OLAP Operations on Data Cube

## Slicing



## Rollup & Drill down





# PIVOT operation in SQL

color attribute를 pivot으로 하고 sales relation을 재배치

| 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        |
| shirt     | dark   | large        | 6        |
| shirt     | pastel | small        | 4        |
| shirt     | pastel | medium       | 1        |
| shirt     | pastel | large        | 2        |
| shirt     | white  | small        | 17       |
| shirt     | white  | medium       | 1        |
| shirt     | white  | large        | 10       |
| pant      | dark   | small        | 14       |
| pant      | dark   | medium       | 6        |
| pant      | dark   | large        | 0        |
| pant      | pastel | small        | 1        |
| pant      | pastel | medium       | 0        |
| pant      | pastel | large        | 1        |
| pant      | white  | small        | 3        |
| pant      | white  | medium       | 0        |
| pant      | white  | large        | 2        |

```
select *
from sales
pivot (
 sum(quantity)
 for color in ('dark','pastel','white')
)
order by item_name;
```



| 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    |
| pants     | small        | 14   | 1      | 3     |
| pants     | medium       | 6    | 0      | 0     |
| pants     | large        | 0    | 1      | 2     |



# Cube operation in SQL: Union of group by's on every subset of the specified attributes

- sales(item\_name, color, clothes\_size, quantity)
- E.g. consider the query

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

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

{ (item\_name, color, clothes\_size), # 3 dimension attributes  
(item\_name, color), (item\_name, clothes\_size), (color, clothes\_size), # 2 dimension attributes  
(item\_name), (color), (clothes\_size), # 1 dimension attribute  
() } # 0 dimension attribute where () denotes an empty **group by** list.

- For each grouping, the result contains the **null** value for attributes not present in the grouping.
- The resulting cube relation will have 80 cells from 4 items, 3 colors, 3 sizes data
  - 4 items, 3 colors, 3 sizes need  $(4+1) \times (3+1) \times (3+1) = 80$  cells in the data cube
- **Union of group by's on every subset of the specified attributes**
- 위 SQL Query 결과물은 80 cells



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

총 80 tuples

item\_name, color, clothes\_size, sum(quantity)  
Item\_name 4가지, Color 3가지, Clothes\_size 3가지  
총 36 combinations → 36 tuples

Item\_name, color, "all", sum(quantity)  
Item\_name 4가지, Color 3가지, Clothes\_size "all"  
총 12 combinations → 12 tuples

Item\_name, "all", clothes\_size, sum(quantity)  
Item\_name 4가지, Color "all", Clothes\_size 3가지  
총 12 combinations → 12 tuples

"all", color, clothes\_size, sum(quantity)  
Item\_name "all", Color 3가지, Clothes\_size 3가지  
총 12 combinations → 9 tuples

Item\_name, "all", "all", sum(quantity)  
Item\_name 4가지, Color "all", Clothes\_size "all"  
총 12 combinations → 4 tuples

"all", color, "all", sum(quantity)  
Item\_name "all", Color 3가지, Clothes\_size "all"  
총 4 combinations → 3 tuples

"all", "all", clothes\_size, sum(quantity)  
Item\_name "all", Color "all", Clothes\_size 3가지  
총 4 combinations → 3 tuples

"all", "all", "all", sum(quantity)  
Item\_name "all", Color "all", Clothes\_size "all"  
총 1 combination → 1 tuple

Original 36 tuples in sales relation

+

|       |        |     |    |
|-------|--------|-----|----|
| skirt | dark   | all | 8  |
| skirt | pastel | all | 35 |
| skirt | white  | all | 10 |
| dress | dark   | all | 20 |
| dress | pastel | all | 10 |
| dress | white  | all | 5  |
| shirt | dark   | all | 14 |
| shirt | pastel | all | 7  |
| shirt | White  | all | 28 |
| pant  | dark   | all | 20 |
| pant  | pastel | all | 2  |
| pant  | white  | all | 5  |

|       |     |     |    |
|-------|-----|-----|----|
| skirt | all | all | 53 |
| dress | all | all | 35 |
| shirt | all | all | 49 |
| pant  | all | all | 27 |

|     |     |     |     |
|-----|-----|-----|-----|
| all | all | all | 164 |
|-----|-----|-----|-----|



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

총 20 tuples

Item\_name, color, "all", sum(quantity)  
 Item\_name 4가지, Color 3가지, Clothes\_size "all"  
 총 12 combinations → 12 tuples

Item\_name, "all", "all", sum(quantity)  
 Item\_name 4가지, Color "all", Clothes\_size "all"  
 총 12 combinations → 4 tuples

"all", color, "all", sum(quantity)  
 Item\_name "all", Color 3가지, Clothes\_size "all"  
 총 4 combinations → 3 tuples

"all", "all", "all", sum(quantity)  
 Item\_name "all", Color "all", Clothes\_size "all"  
 총 1 combination → 1 tuple

| 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      |



# Roll Up [1/2]

## Union on every prefix of specified attributes

- E.g., `select item_name, color, clothes_size, sum(quantity)`  
`from sales`  
`group by rollup(item_name, color, clothes_size)`

총 36+17=  
53 tuples

- Generates union of 4 groupings of every prefix of specified attributes:  
 $\{ (item\_name, color, clothes\_size), (item\_name, color), (item\_name), () \}$

| 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        |
| shirt     | dark   | large        | 6        |
| shirt     | pastel | small        | 4        |
| shirt     | pastel | medium       | 1        |
| shirt     | pastel | large        | 2        |
| shirt     | white  | small        | 17       |
| shirt     | white  | medium       | 1        |
| shirt     | white  | large        | 10       |
| pant      | dark   | small        | 14       |
| pant      | dark   | medium       | 6        |
| pant      | dark   | large        | 0        |
| pant      | pastel | small        | 1        |
| pant      | pastel | medium       | 0        |
| pant      | pastel | large        | 1        |
| pant      | white  | small        | 3        |
| pant      | white  | medium       | 0        |
| pant      | white  | large        | 2        |

`item_name, color, clothes_size, sum(quantity)`  
Item\_name 4가지, Color 3가지, Clothes\_size 3가지  
총 36 combinations → 36 tuples



`Item_name, color, "all", sum(quantity)`  
Item\_name 4가지, Color 3가지, Clothes\_size "all"  
총 12 combinations → 12 tuples



`Item_name, "all", "all", sum(quantity)`  
Item\_name 4가지, Color "all", Clothes\_size "all"  
총 4 combinations → 4 tuples



`"all", "all", "all", sum(quantity)`  
Item\_name "all", Color "all", Clothes\_size "all"  
총 1 combination → 1 tuple

Original 36 tuples



|       |        |     |    |
|-------|--------|-----|----|
| skirt | dark   | all | 8  |
| skirt | pastel | all | 35 |
| skirt | white  | all | 10 |
| dress | dark   | all | 20 |
| dress | pastel | all | 10 |
| dress | white  | all | 5  |

|       |        |     |    |
|-------|--------|-----|----|
| shirt | dark   | all | 14 |
| shirt | pastel | all | 7  |
| shirt | White  | all | 28 |

|      |        |     |    |
|------|--------|-----|----|
| pant | dark   | all | 20 |
| pant | pastel | all | 2  |
| pant | white  | all | 5  |

|       |     |     |    |
|-------|-----|-----|----|
| skirt | all | all | 53 |
| dress | all | all | 35 |
| shirt | all | all | 49 |
| pant  | all | all | 27 |

|     |     |     |     |
|-----|-----|-----|-----|
| all | all | all | 164 |
|-----|-----|-----|-----|



# Roll Up [2/2]

- E.g., `select item_name, color, sum(quantity)`  
`from sales`  
`group by rollup(item_name, color)`
- Generates union of 3 groupings of every prefix of specified attributes:  
 $\{ (item\_name, color), (item\_name), () \}$

| 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        |
| shirt     | dark   | large        | 6        |
| shirt     | pastel | small        | 4        |
| shirt     | pastel | medium       | 1        |
| shirt     | pastel | large        | 2        |
| shirt     | white  | small        | 17       |
| shirt     | white  | medium       | 1        |
| shirt     | white  | large        | 10       |
| pant      | dark   | small        | 14       |
| pant      | dark   | medium       | 6        |
| pant      | dark   | large        | 0        |
| pant      | pastel | small        | 1        |
| pant      | pastel | medium       | 0        |
| pant      | pastel | large        | 1        |
| pant      | white  | small        | 3        |
| pant      | white  | medium       | 0        |
| pant      | white  | large        | 2        |

Item\_name, color, sum(quantity)  
 Item\_name 4가지, Color 3가지, Clothes\_size “all”  
 총 12 combinations → 12 tuples

Item\_name, “all”, sum(quantity)  
 Item\_name 4가지, Color “all”, Clothes\_size “all”  
 총 4 combinations → 4 tuples

“all”, “all”, sum(quantity)  
 Item\_name “all”, Color “all”, Clothes\_size “all”  
 총 1 combination → 1 tuple

총 17 tuples

|       |        |     |
|-------|--------|-----|
| 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   |
| skirt | all    | 53  |
| dress | all    | 35  |
| shirt | all    | 49  |
| pant  | all    | 27  |
| all   | all    | 164 |



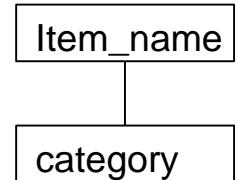
# Roll Up with Hierarchy

- E.g., suppose we have a table `itemcategory(category, item_name)` 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*

| (category, item_name) |
|-----------------------|
| wowenswear, skirt     |
| wowenswear, dress     |
| menswear, pants       |
| menswear, pants       |



Itemcategory hierarchy

|            |       |        |     |
|------------|-------|--------|-----|
| womenswear | skirt | dark   | 8   |
| womenswear | skirt | pastel | 35  |
| womenswear | skirt | white  | 10  |
| womenswear | dress | dark   | 20  |
| womenswear | dress | pastel | 10  |
| womenswear | dress | white  | 5   |
| menswear   | shirt | dark   | 14  |
| menswear   | shirt | pastel | 7   |
| menswear   | shirt | White  | 28  |
| menswear   | pant  | dark   | 20  |
| menswear   | pant  | pastel | 2   |
| menswear   | pant  | white  | 5   |
| womenswear | skirt | all    | 53  |
| womenswear | dress | all    | 35  |
| womenswear | shirt | all    | 49  |
| womenswear | pant  | all    | 27  |
| womenswear | all   | all    | 164 |
| menswear   | skirt | all    | 53  |
| menswear   | dress | all    | 35  |
| menswear   | shirt | all    | 49  |
| menswear   | pant  | all    | 27  |
| menswear   | all   | all    | 164 |

We can easily get this!

| clothes_size: all |           |       |        |       |       |     |
|-------------------|-----------|-------|--------|-------|-------|-----|
| category          | item_name | color |        |       |       |     |
|                   |           | dark  | pastel | white | total |     |
| womenswear        | skirt     | 8     | 8      | 35    | 53    |     |
|                   | dress     | 20    | 20     | 10    | 35    |     |
|                   | subtotal  | 28    | 28     | 45    |       | 88  |
| menswear          | pants     | 14    | 14     | 7     | 49    |     |
|                   | shirt     | 20    | 20     | 2     | 27    |     |
|                   | subtotal  | 34    | 34     | 9     |       | 76  |
| total             |           | 62    | 62     | 54    |       | 164 |



# Summary Table using Cube & Rollup

Sales Relation

| Shop<br>Addr | State | Year | amt_<br>paid |
|--------------|-------|------|--------------|
| LA           | CA    | 2011 | 300          |
| SF           | CA    | 2011 | 200          |
| LA           | CA    | 2012 | 350          |
| SD           | CA    | 2011 | 130          |
| NY           | NY    | 2012 | 300          |
| Queens       | NY    | 2012 | 200          |
| ...          | ....  | .... | .....        |

## Rollup

```
select state, year, sum(amt_paid)
from sales
group by rollup(state, year)
```

| State | Year | Sum   |
|-------|------|-------|
| CA    | 2011 | 20000 |
| CA    | 2012 | 25000 |
| CA    | *    | 45000 |
| NY    | 2012 | 15000 |
| NY    | *    | 15000 |
| *     | *    | 60000 |

{State, Year} {State} {} 를  
subset으로 생성

## Cube

```
select state, year, sum(amt_paid)
from sales
group by cube(state, year)
```

| State | Year | Sum   |
|-------|------|-------|
| CA    | 2011 | 20000 |
| CA    | 2012 | 25000 |
| CA    | *    | 45000 |
| NY    | 2012 | 15000 |
| NY    | *    | 15000 |
| *     | *    | 60000 |
| *     | 2011 | 20000 |
| *     | 2012 | 40000 |

{State, Year} {State} {Year} {} 를  
subset으로 생성



# Relation with “null” to Cross-tab with “all”

- Relational representation of cross-tab (in next slide) 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)
```

- 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)**



# Multiple Rollups & Cubes

- 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

$$\begin{aligned} & \{item\_name, ()\} \times \{(color, size), (color), ()\} \\ &= \{ (item\_name, color, size), (item\_name, color), (item\_name), \\ & \quad (color, size), (color), () \} \end{aligned}$$



**Figure 5.22: Result of SQL pivot operation on sales relation of Fig 5.16**

| 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: Query Result**

| item_name | quantity |
|-----------|----------|
| skirt     | 53       |
| dress     | 35       |
| shirt     | 49       |
| pant      | 27       |

**Figure 5.24: Query Result**

| 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        |



# 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
- Early OLAP systems precomputed ***all possible aggregates*** in order to provide online response
  - Space and time requirements for doing so can be very high
    - ▶ Cube operation needs  $2^n$  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  $(item\_name, color, size)$
    - ▶ Can compute aggregates on  $(item\_name, color, size)$ ,  $(item\_name, color)$  and  $(item\_name)$  using a single sorting of the base data