DATABASE SYSTEM PRINCIPLE – ADVANCED SQL

李旭东

LEEXUDONG@NANKAI.EDU.CN

NANKAI UNIVERSITY

OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

ACCESSING SQL FROM A PROGRAMMING LANGUAGE

- 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

ACCESSING SQL FROM A PROGRAMMING LANGUAGE

- Various tools:
  - JDBC (Java Database Connectivity) works with Java
  - ODBC (Open Database Connectivity) works with C, C++, C#, …
  - Embedded SQL

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.

JDBC

- 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: JAVA VM + ECLIPSE IDE + MYSQL DB

- JAVA VM(Java Platform (JDK) 8u121):

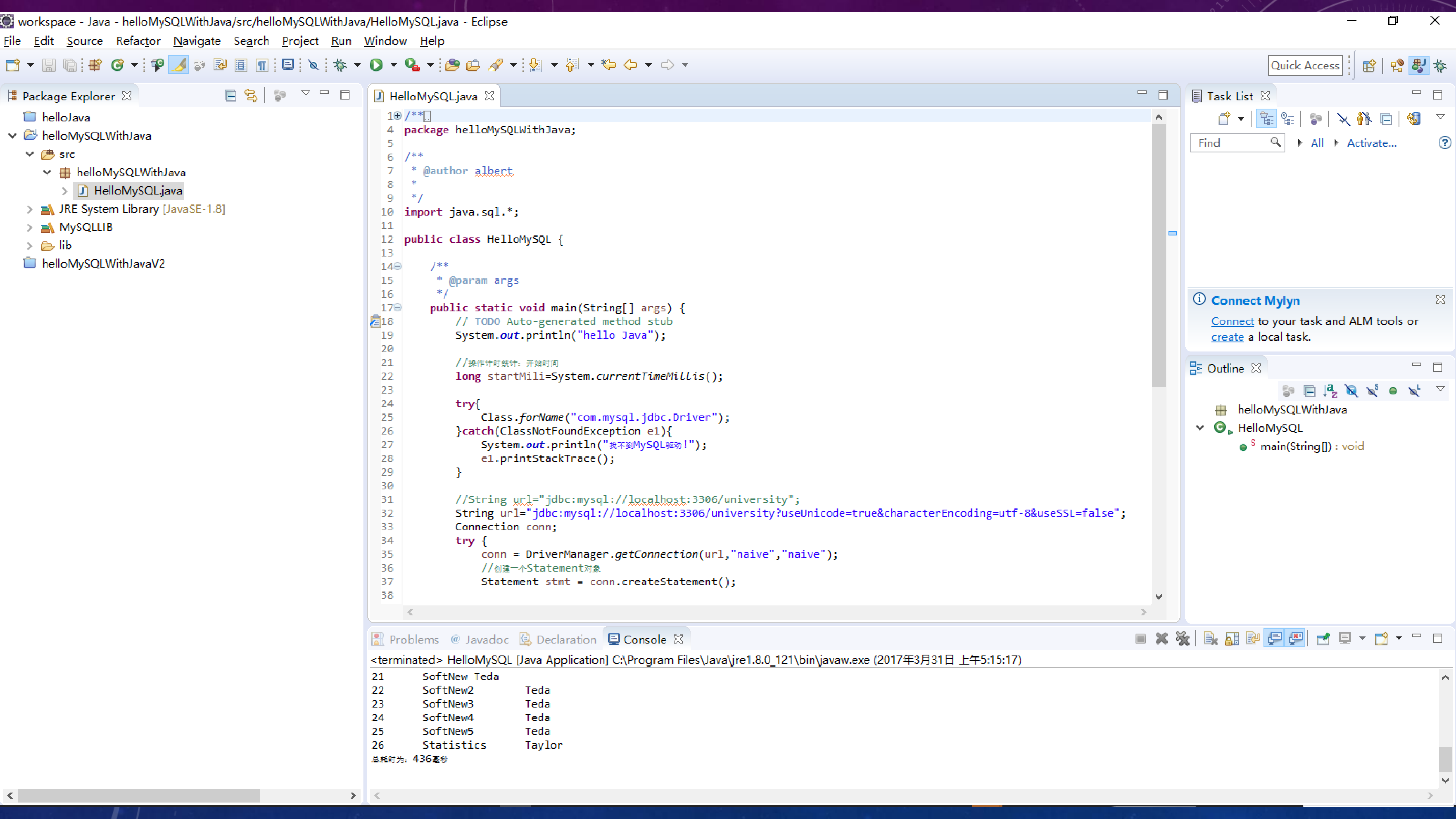
http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html

- eclipse IDE(Eclipse Neon):

http://www.eclipse.org/downloads/

- MySQL Connector for Java (mysql-connector-java-5.1.41.zip):

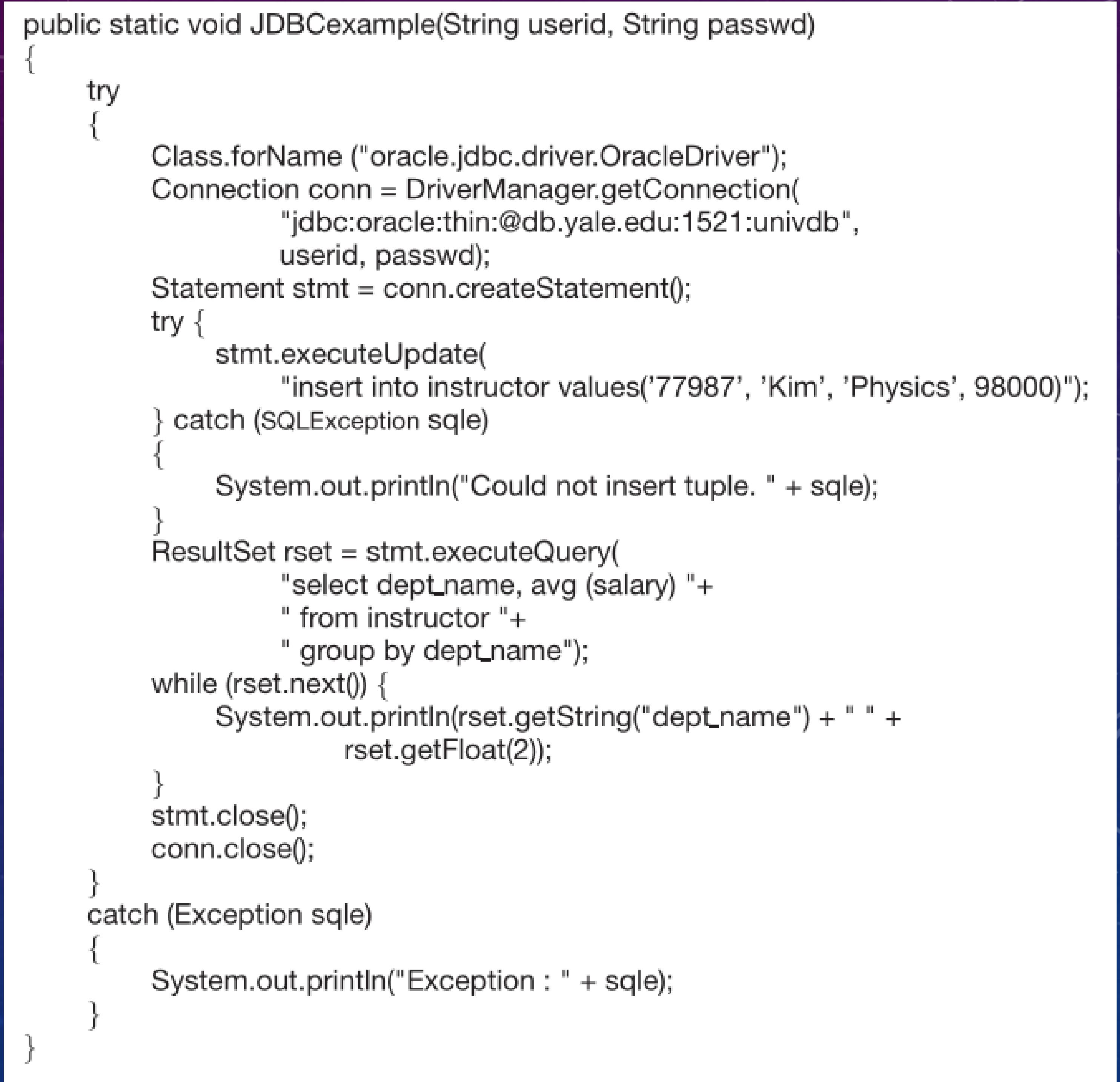
https://dev.mysql.com/downloads/connector/j/


workspace - Java - helloMySQLWithJava/src/helloMySQLWithJava/HelloMySQL.java - Eclipse

```java
/**
package helloMySQLWithJava;

/**
 * @author albert
 *
 */
import java.sql.*;

public class HelloMySQL {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("hello Java");

        //操作计时统计：开始时间
        long startMili=System.currentTimeMillis();

        try{
            Class.forName("com.mysql.jdbc.Driver");
        }catch(ClassNotFoundException e1){
            System.out.println("找不到MySQL驱动！");
            e1.printStackTrace();
        }

        //String url="jdbc:mysql://localhost:3306/university";
        String url="jdbc:mysql://localhost:3306/university?useUnicode=true&characterEncoding=utf-8&useSSL=false";
        Connection conn;
        try {
            conn = DriverManager.getConnection(url,"naive","naive");
            //创建一个Statement对象
            Statement stmt = conn.createStatement();

```

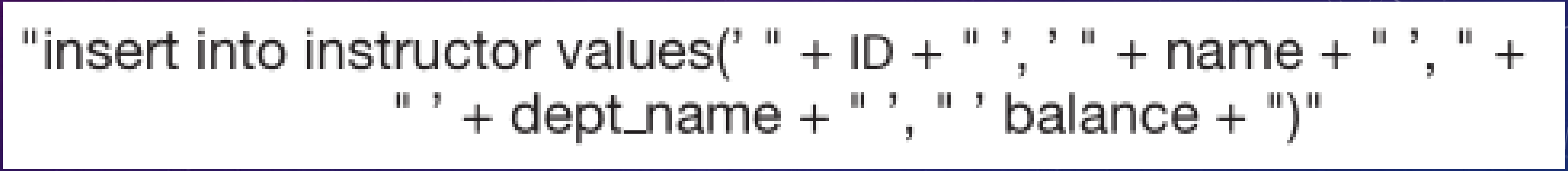
```
<terminated> HelloMySQL [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (2017年3月31日 上午5:15:17)
21      SoftNew Teda
22      SoftNew2        Teda
23      SoftNew3        Teda
24      SoftNew4        Teda
25      SoftNew5        Teda
26      Statistics      Taylor
总耗时为：436毫秒
```

JDBC



```
public static void JDBCexample(String userid, String passwd)
{
    try
    {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                userid, passwd);
        Statement stmt = conn.createStatement();
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
        } catch (SQLException sqle)
        {
            System.out.println("Could not insert tuple. " + sqle);
        }
        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));
        }
        stmt.close();
        conn.close();
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

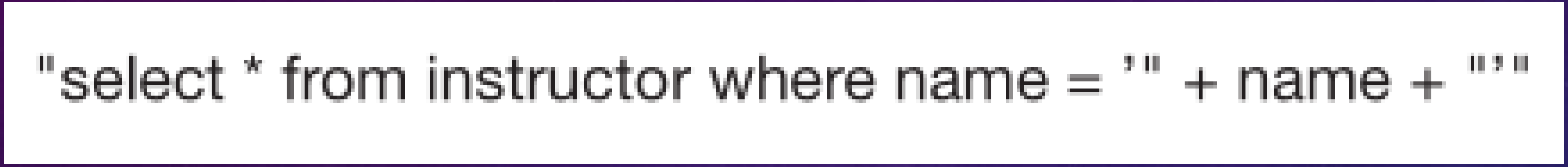
JDBC



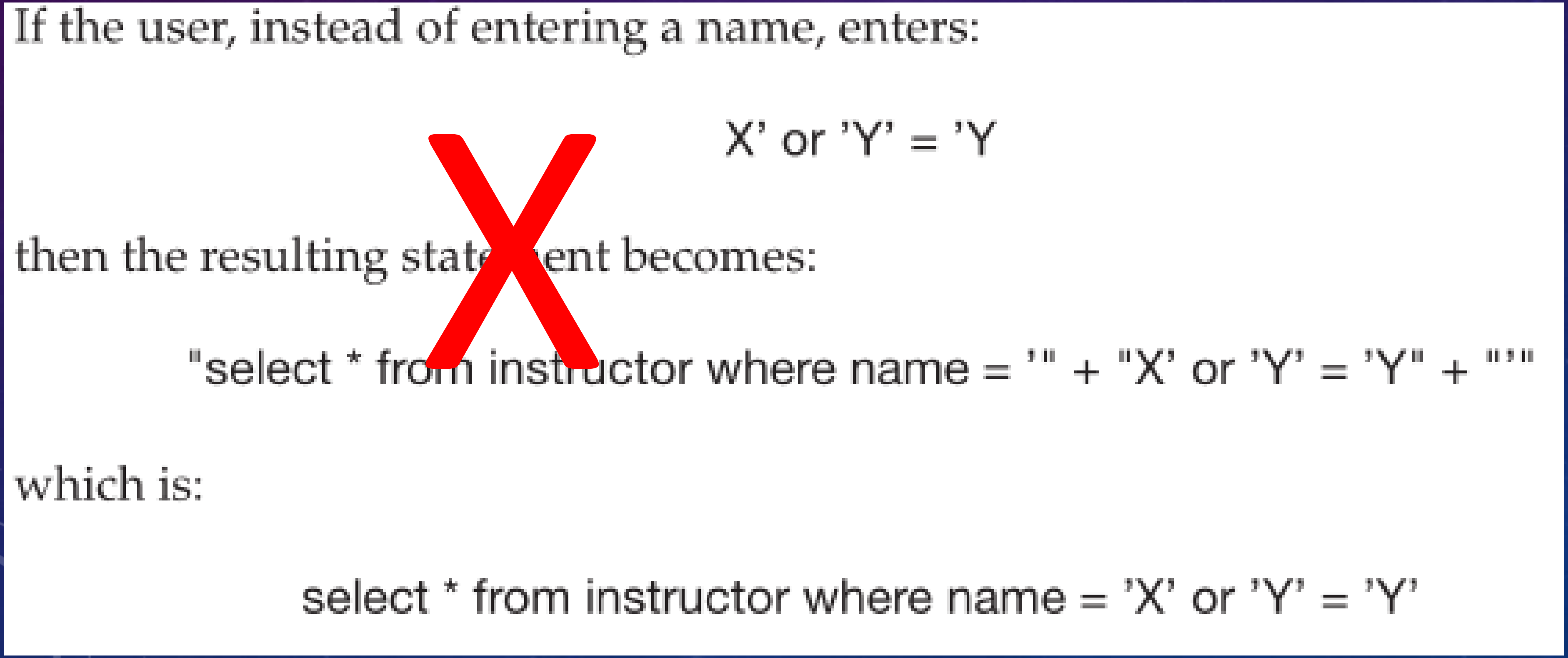
```
"insert into instructor values(' " + ID + " ', ' " + name + " ', " +
                    " ' + dept_name + " ', " ' balance + ")"
```

©LXD

JDBC: SQL INJECTION



```
"select * from instructor where name = ’" + name + "’"
```



If the user, instead of entering a name, enters:

```
X’ or ’Y’ = ’Y
```

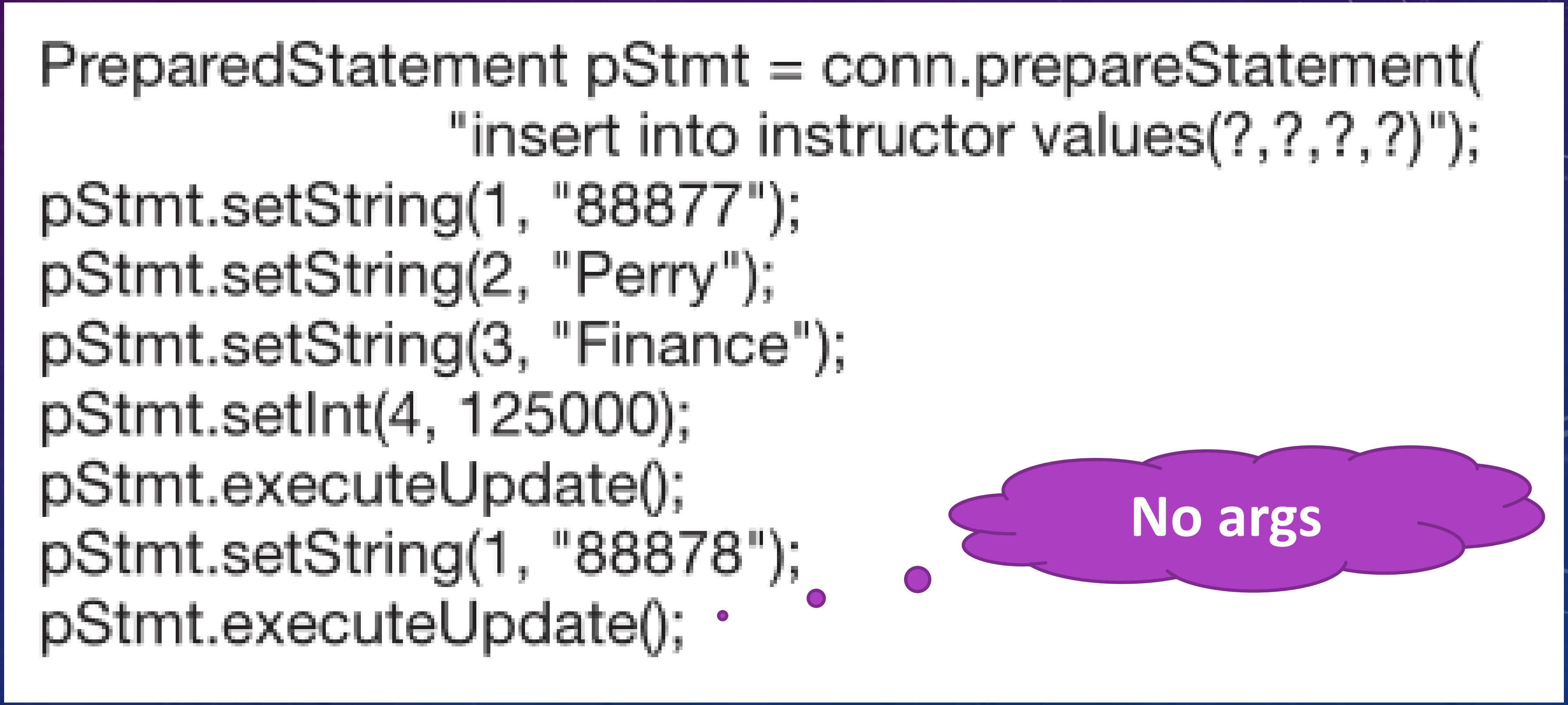
then the resulting statement becomes:

```
"select * from instructor where name = ’" + "X’ or ’Y’ = ’Y" + "’"
```

which is:

```
select * from instructor where name = ’X’ or ’Y’ = ’Y’
```

JDBC: PREPARED STATEMENTS



```
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();
```

No args

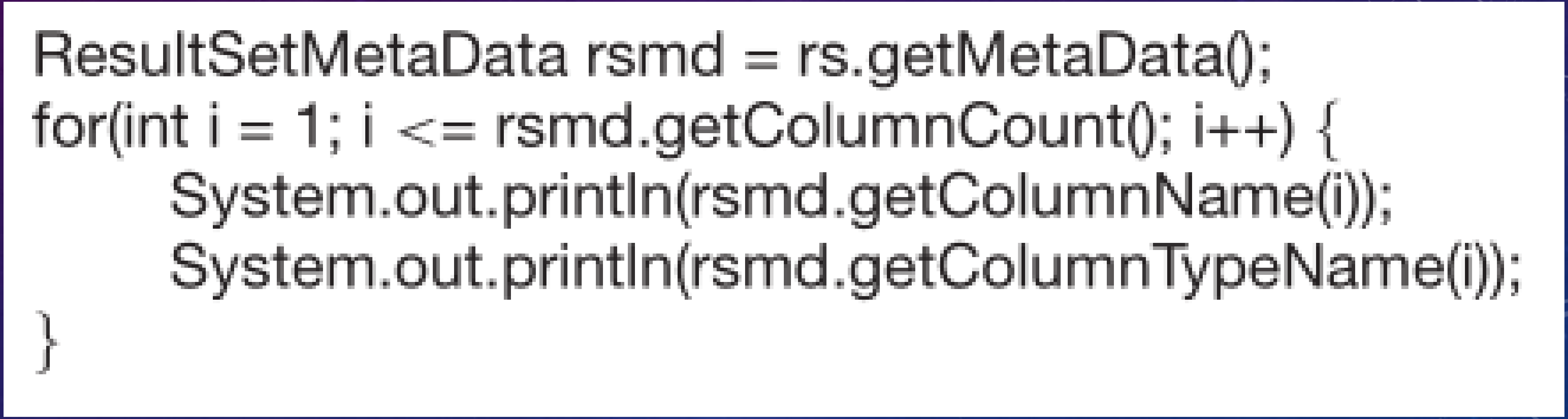
JDBC: PREPARED STATEMENTS

- Blob, Clob
- PreparedStatement
    - setBlob(int parameterIndex, InputStream inputStream)
    - getBlob()

JDBC：CALLABLE STATEMENTS 可调用语句

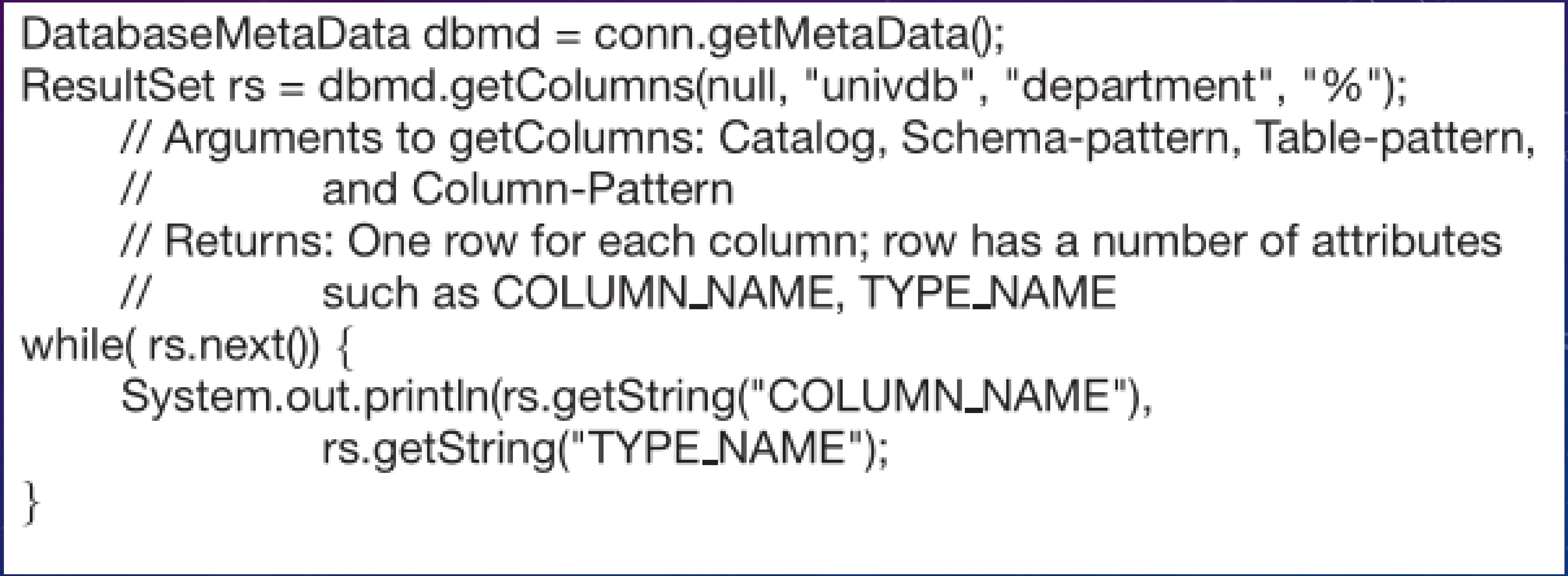
```
String sql = "call sp_department1(?,?);";
CallableStatement  cs = null;
cs =  conn.prepareCall(sql);
cs.setString(1, "Soft");
cs.registerOutParameter(2, Types.INTEGER) ;
cs.execute() ;
int mCnt = cs.getInt(2);
cs.close();
```

JDBC:
METADATA FEATURES



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

JDBC:
METADATA FEATURES

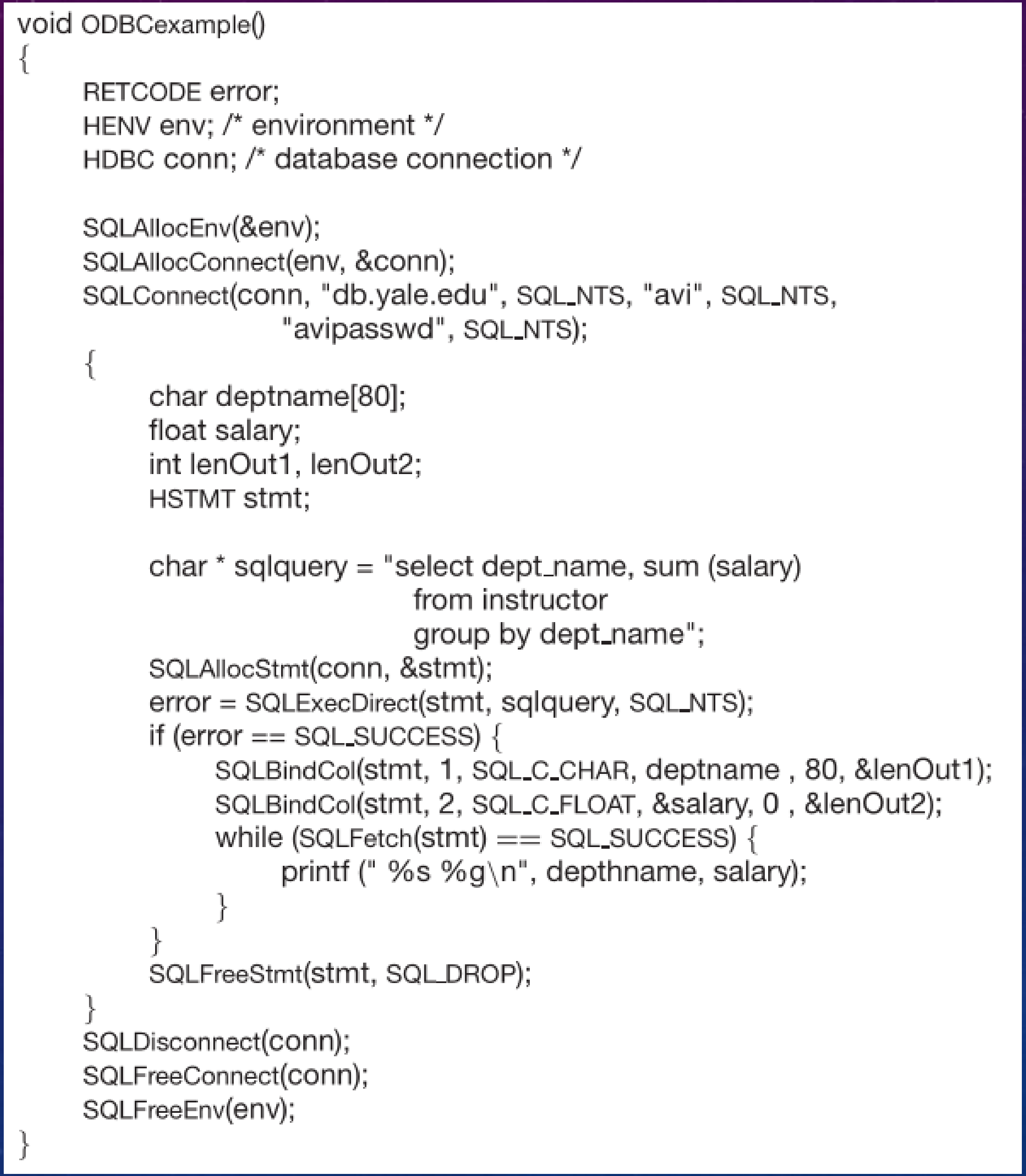


```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
        // Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
        //                 and Column-Pattern
        // Returns: One row for each column; row has a number of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

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

ODBC



```
void 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);
      {
            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", depthname, salary);
                  }
            }
            SQLFreeStmt(stmt, SQL_DROP);
      }
      SQLDisconnect(conn);
      SQLFreeConnect(conn);
      SQLFreeEnv(env);
}
```

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.

EMBEDDED SQL

- **EXEC SQL** statement is used 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

- 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*)

EMBEDDED SQL

- 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

- 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

EMBEDDED SQL

- Example: (cont.,)
  - Specify the query in SQL as follows:

```
EXEC SQL
  declare c cursor for
  select ID, name
  from student
  where tot_cred > :credit_amount
END_EXEC
```

EMBEDDED SQL

- 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* END_EXEC

  Repeated calls to fetch get successive tuples in the query result

EMBEDDED SQL

- 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 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

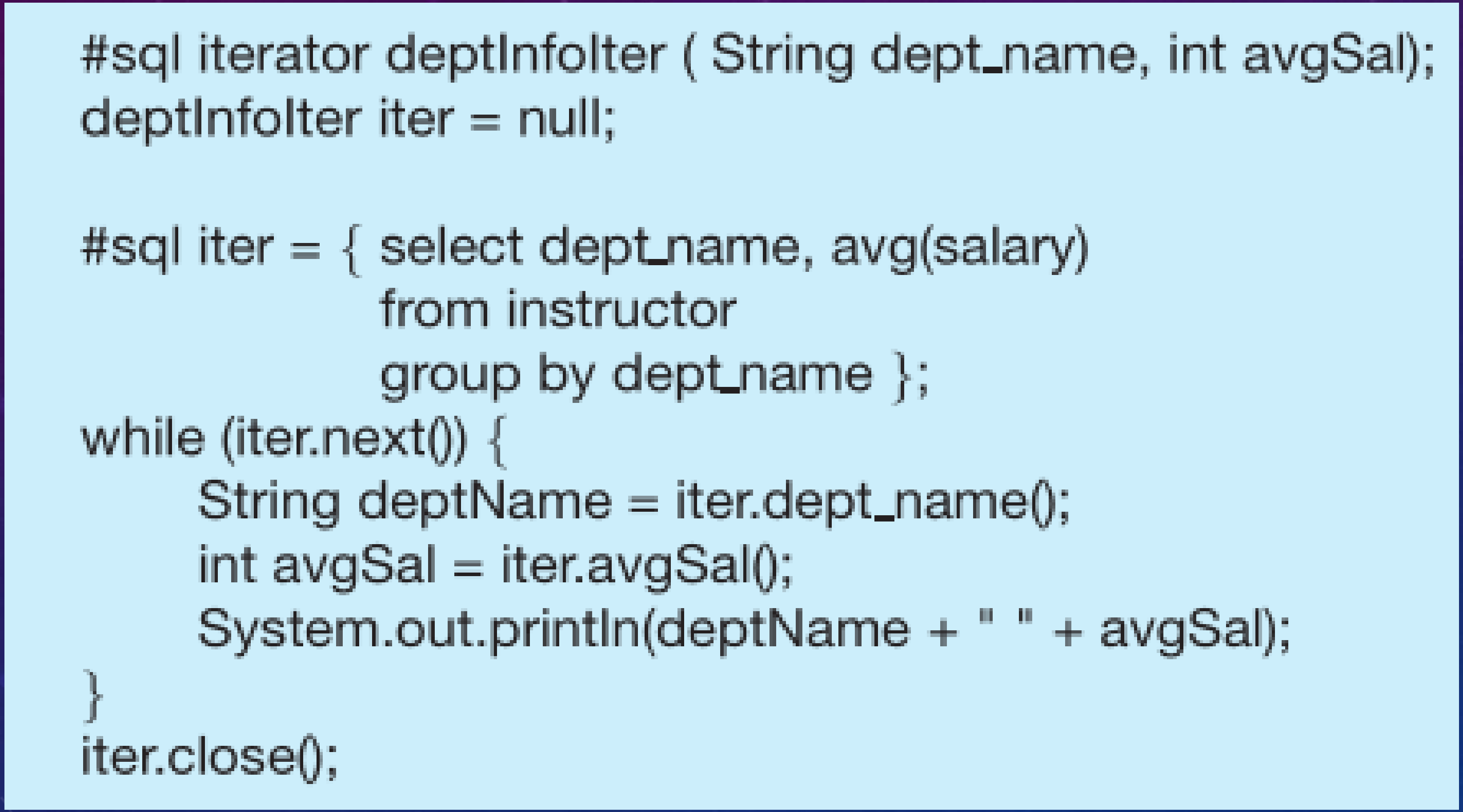
```
EXEC SQL
     declare c cursor for
    select *
    from instructor
    where dept_name = ‘Music’
    for update
```

UPDATES THROUGH EMBEDDED SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)
- We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

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

SQLJ



```
#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();
```

PYTHON

```
import pymysql
conn=pymysql.connect(host='localhost',user=‘user',password=‘pwd',database='schema1',charset='utf8')
cursor=conn.cursor()
sql = "select name,salary,dept_name from instructor where dept_name like '%s'" %("C"+"%")
cursor.execute(sql)
retdat = cursor.fetchall()
for row in retdat:
    print(row[0], ':', row[1], ':', row[2])
cursor.close()
conn.close()
```

Pymysql error： cryptography is required for sha256_password or caching_sha2_password
pip install cryptography

OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

TEMPORARY TABLE

- Temporary Table
  - a temporary table is a special type of table that allows you to store a temporary result set, which you can reuse several times in a single session
  - MySQL removes the temporary table automatically when the session ends or the connection is terminated
  - A temporary table can have the same name as a normal table in a database. For example, if you create a temporary table named employees in the sample database, the existing employees table becomes inaccessible.

TEMPORARY TABLE

- Temporary Table
  - A temporary table is only available and accessible to the client that creates it.
  - Different clients can create temporary tables with the same name without causing errors because only the client that creates the temporary table can see it.
  - However, in the same session, two temporary tables cannot share the same name.

TEMPORARY TABLE – CREATE

```
CREATE TEMPORARY TABLE top10customers
SELECT p.customerNumber,  c.customerName,
       ROUND(SUM(p.amount), 2)  sales
FROM payments p
INNER JOIN customers c ON c.customerNumber = p.customerNumber
GROUP BY p.customerNumber
ORDER BY sales DESC
LIMIT 10;
```

round(x,d) :四舍五入
x指要处理的数,
d是指保留几位小数

TEMPORARY TABLE – RETRIEVE

```
SELECT
    customerNumber,
    customerName,
    sales
FROM
    top10customers
ORDER BY sales;
```

TEMPORARY TABLE – DROP

```
DROP TEMPORARY TABLE top10customers;
```

OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

FUNCTIONS AND PROCEDURES

- SQL:1999 supports functions and procedures
  - Functions/procedures can be written in SQL itself, or in an external programming language
  - Functions written in an external languages 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.

FUNCTIONS AND PROCEDURES

- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.
- PL/SQL (oracle), TransactSQL (ms sql server), PL/pgSQL

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
```

SQL FUNCTIONS(CONT.,)

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.
- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

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

SQL FUNCTIONS

- Compound statement: **begin … end**
  - May contain multiple SQL statements between **begin** and **end**.
- **returns** -- indicates the variable-type that is returned (e.g., integer)
- **return** **--** specifies the values that are to be returned as result of invoking the function
- SQL function are in fact parameterized views that generalize the regular notion of views by allowing parameters.

TABLE FUNCTIONS

- SQL:2003 added functions that return a relation as a result
- Example: Return all instructors in a given department

```
create function instructor_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 = instructor_of.dept_name)
```

TABLE FUNCTIONS

Cont.,

- Usage

  select *
  from table (*instructor_of* (‘Music’))

DECLARE FUNCTION IN MYSQL

```
drop function if exists  precourse_cnt_f;
DELIMITER $$
create function precourse_cnt_f(acourseid varchar(8))
returns integer
deterministic
reads sql data
begin
  declare cnt integer;
  select count(*) into cnt from prereq  where course_id=acourseid;
  return cnt;
end$$
DELIMITER ;
```

CALL FUNCTION IN MYSQL

- In sql functions and procedures

```
declare course_id varchar(10);
select ‘958’ into course_id;
select precourse_cnt_f(course_id);
```

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
```

SQL PROCEDURES （CONT.，）

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

  create or replace procedure *dept_count_proc* (**in** *dept_name* **varchar**(20),
  out *d_count* **integer)**

  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 PROCEDURES （CONT.，）

- 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

LANGUAGE CONSTRUCTS FOR PROCEDURES & FUNCTIONS

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
  - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin … end**,
  - May contain multiple SQL statements between **begin** and **end**.
  - Local variables can be declared within a compound statements

LANGUAGE CONSTRUCTS FOR PROCEDURES & FUNCTIONS (CONT.,)

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
- **While** and **repeat** statements:

```
while boolean expression do
        sequence of statements ;
end while
```

```
repeat
        sequence of statements ;
until boolean expression
end repeat
```

LANGUAGE CONSTRUCTS FOR PROCEDURES & FUNCTIONS (CONT.,)

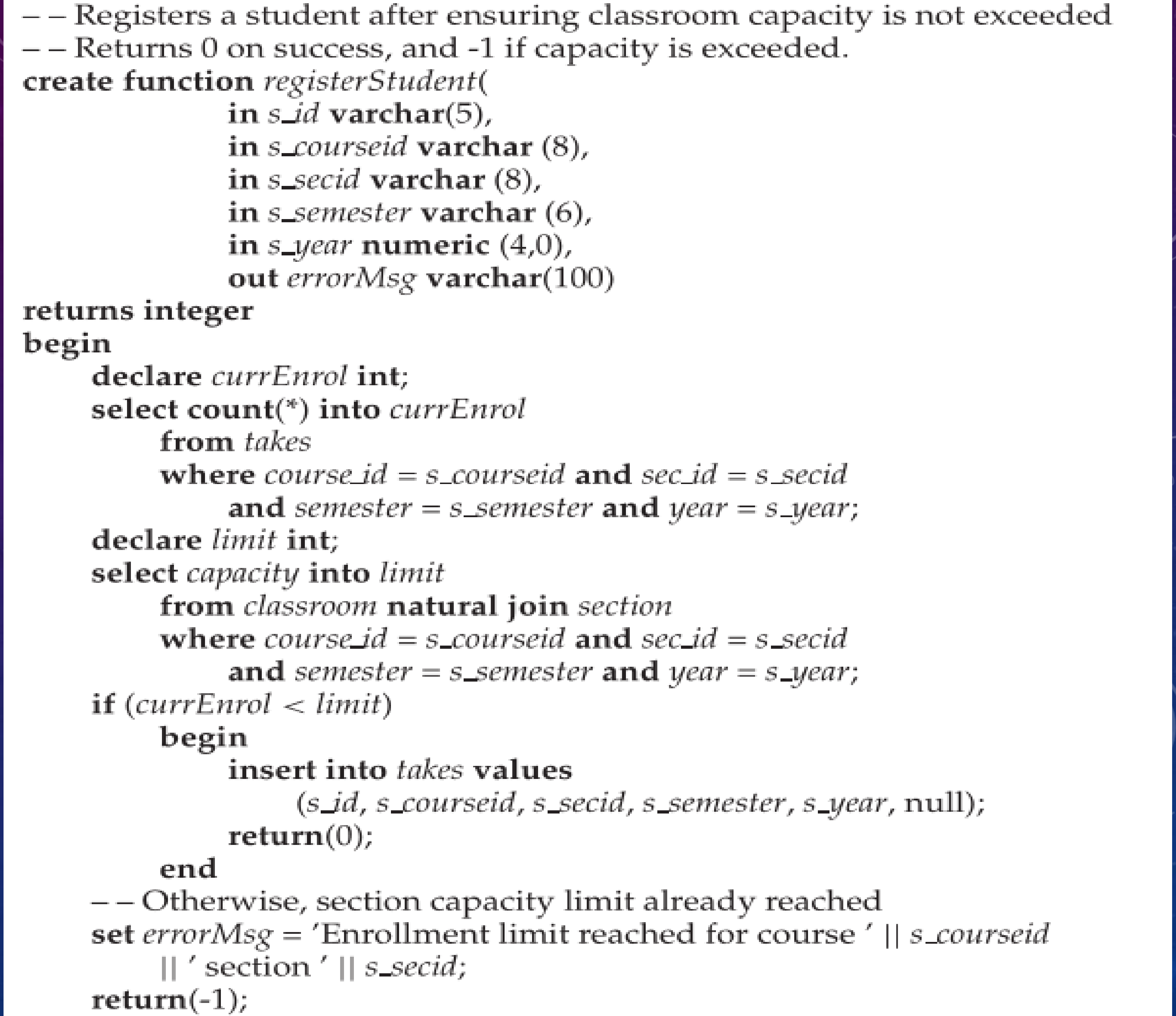
- **For** loop
  - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n  integer default 0;
for r  as
      select budget from department
do
      set n = n + r.budget
end for
```

LANGUAGE CONSTRUCTS FOR PROCEDURES & FUNCTIONS: IF-THEN-ELSE

- Conditional statements (**if-then-else**)
  SQL:1999 also supports a **case** statement similar to C case statement
- Example procedure: registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - See book (page 177) for details

LANGUAGE CONSTRUCTS FOR PROCEDURES & FUNCTIONS : IF-THEN-ELSE



```
– – Registers a 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);
```

©LXD

LANGUAGE CONSTRUCTS FOR PROCEDURES & FUNCTIONS : EXCEPTION CONDITION异常条件

- Signaling of exception conditions, and declaring handlers for exceptions

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

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

LANGUAGE CONSTRUCTS FOR PROCEDURES & FUNCTIONS : EXCEPTION CONDITION异常条件

- declaring handlers for exceptions
  - exit
  - continue
  - sqlexception
  - sqlwarning
  - not found
  - …

EXTERNAL LANGUAGE ROUTINES

- 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.

EXTERNAL LANGUAGE ROUTINES (CONT.,)

- 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.

EXTERNAL LANGUAGE ROUTINES （CONT.，）

- To deal with security problems, we can do on of the following:
  - Use sandbox techniques
    - That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code.
  - 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.

MYSQL PROCEDURE

```
DELIMITER $$
drop procedure if exists sp_department1;
create procedure sp_department1(out cnt integer)
begin  select count(*) into cnt from department;
end$$
DELIMITER ;
```

OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

TRIGGERS触发器

- A trigger is a statement that the system executes automatically as a side effect of a modification to the database.

TRIGGERS

- 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.

TRIGGERING EVENTS AND ACTIONS IN SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
    - For example, **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. For example, convert blank grades to null.

TRIGGERING EVENTS AND ACTIONS IN SQL

```
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

DECLARE TRIGGER IN MYSQL

```
drop trigger if exists  instuctor03_salary;
DELIMITER $$
create trigger instuctor03_salary before update on instructor03
for each row
begin
declare MESSAGE_TEXT varchar(200);
if NEW.salary <0 then
  signal sqlstate 'AL001' set MESSAGE_TEXT = 'error salary is less than zero';
end if;
end$$
DELIMITER ;
```

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

WHEN NOT TO USE TRIGGERS (CONT.)

- 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 (CONT.)

- 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

TRIGGER

- create trigger   XXX
- disable trigger   XXX
- drop trigger   XXX

OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

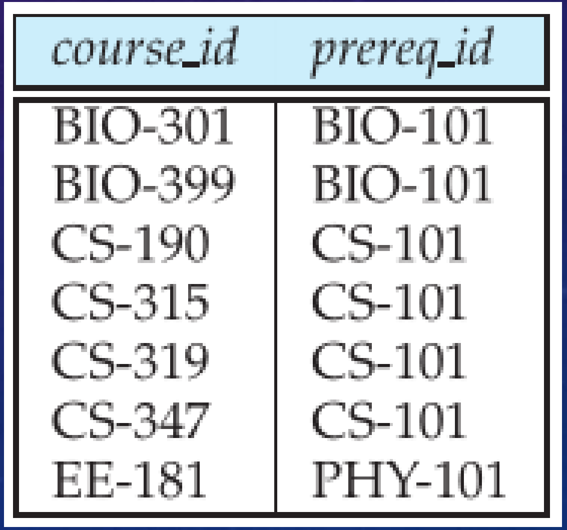
OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

RECURSION IN SQL

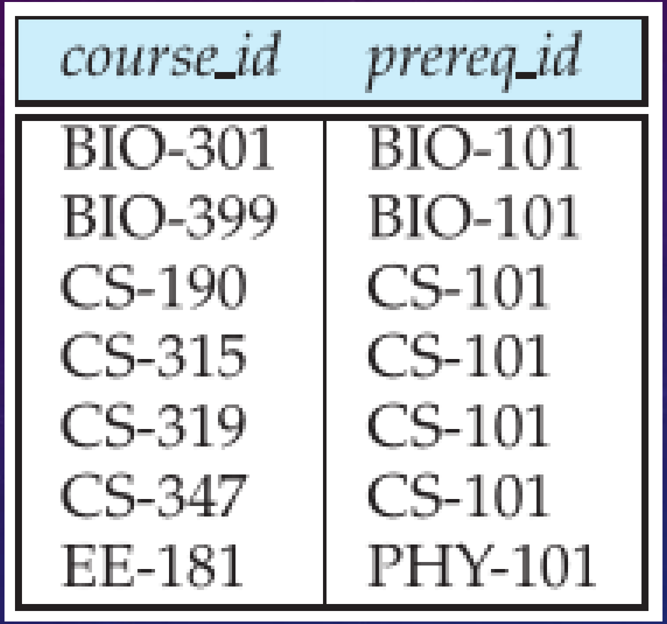
- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
create table prereq    (
    course_id       varchar(8),
    prereq_id       varchar(8),
    primary key (course_id, prereq_id),
    foreign key (course_id) references course(course_id) on delete cascade,
foreign key (prereq_id) references course(course_id)   );
```



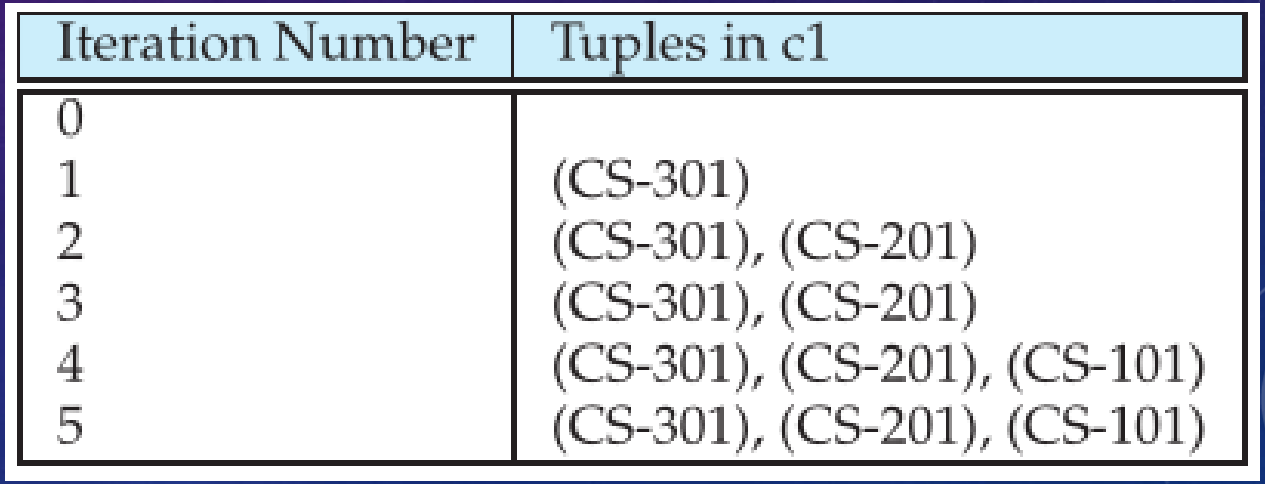
course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

RECURSION IN SQL: TRANSITIVE CLOSURE传递闭包 USING ITERATION 1/3



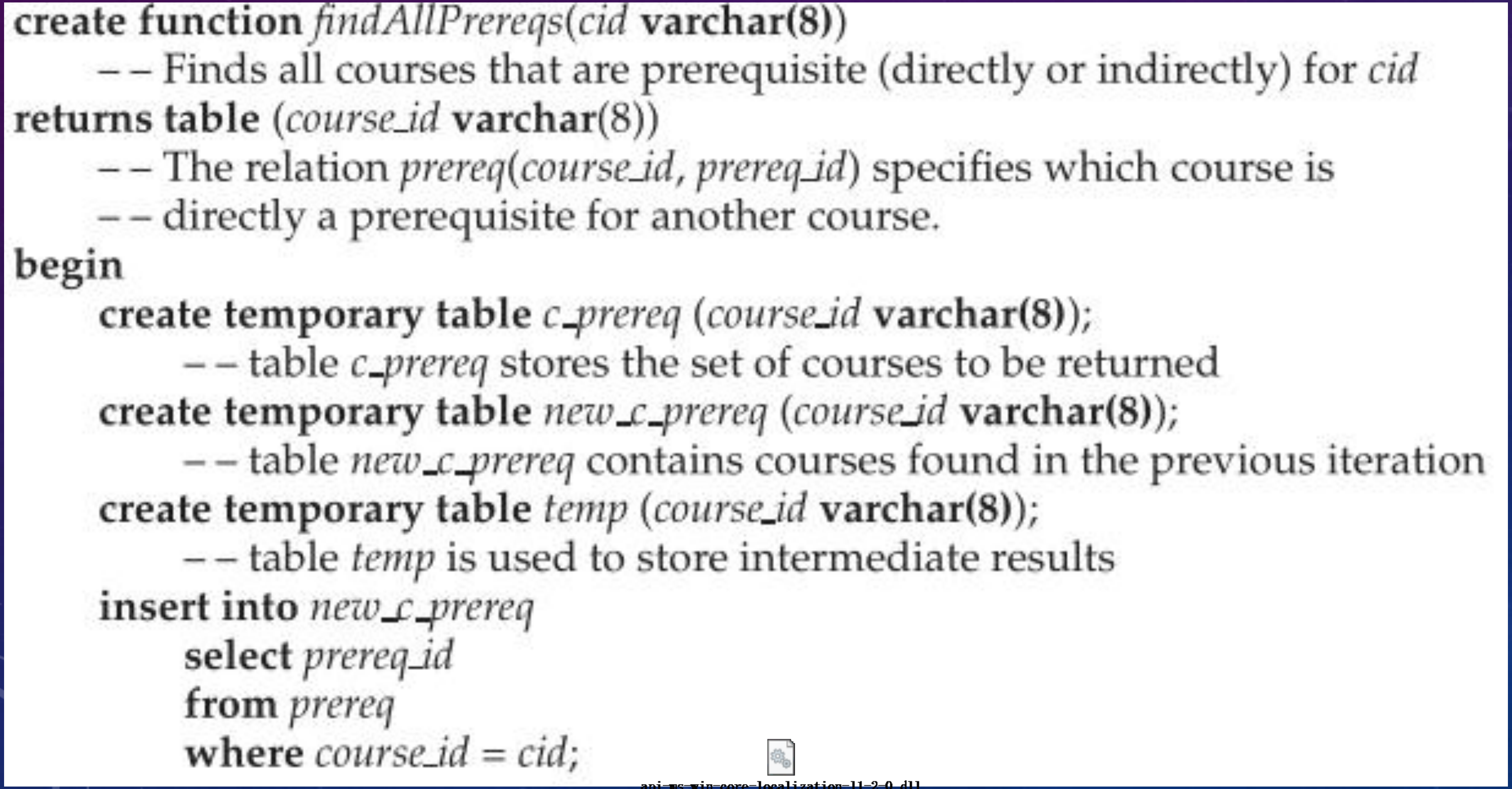
course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

CS-347



Iteration Number	Tuples in c1
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

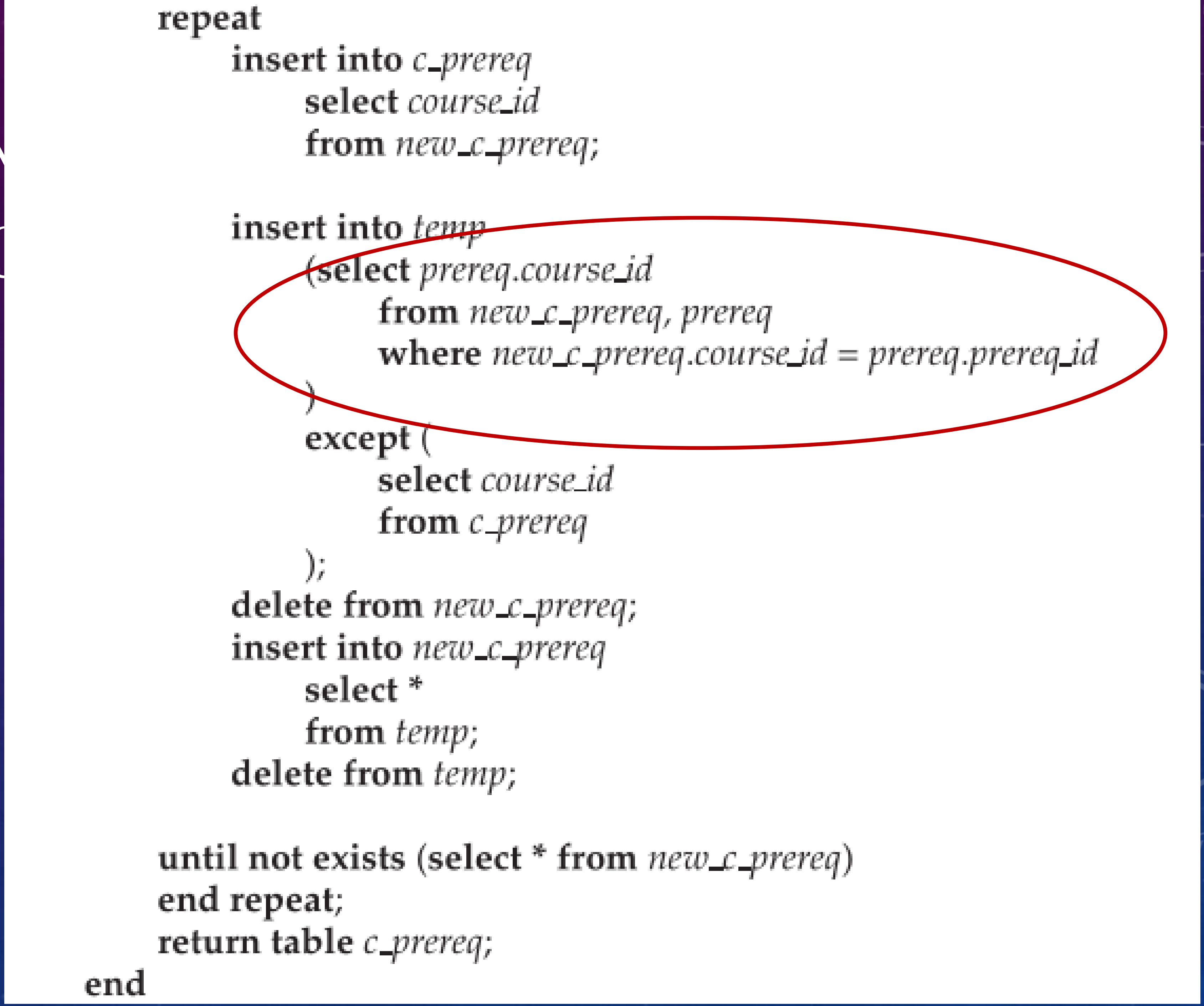
RECURSION IN SQL: TRANSITIVE CLOSURE USING ITERATION 2/3



```
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;
```

api-ms-win-core-localization-l1-2-0.dll

RECURSION IN TRANSITIVE C



```
    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
```

RECURSION IN SQL

```
• 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;
```

OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

ADVANCED AGGREGATION FEATURES: RANK排名

- 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*

ADVANCED AGGREGATION FEATURES: RANKING排名

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

  student_grades(ID, GPA)
- 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

ADVANCED AGGREGATION FEATURES: 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;
```

ADVANCED AGGREGATION FEATURES: RANKING (PARTITION)

- 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.

ADVANCED AGGREGATION FEATURES: RANKING (PARTITION)

(Cont.,)

- 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

ADVANCED AGGREGATION FEATURES: RANKING

- 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*

ADVANCED AGGREGATION FEATURES: RANKING

- 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;*

ADVANCED AGGREGATION FEATURES: WINDOWING分窗

- 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"移动平均
- **Window specification** in SQL:
  - Given relation *sales(date, value)*

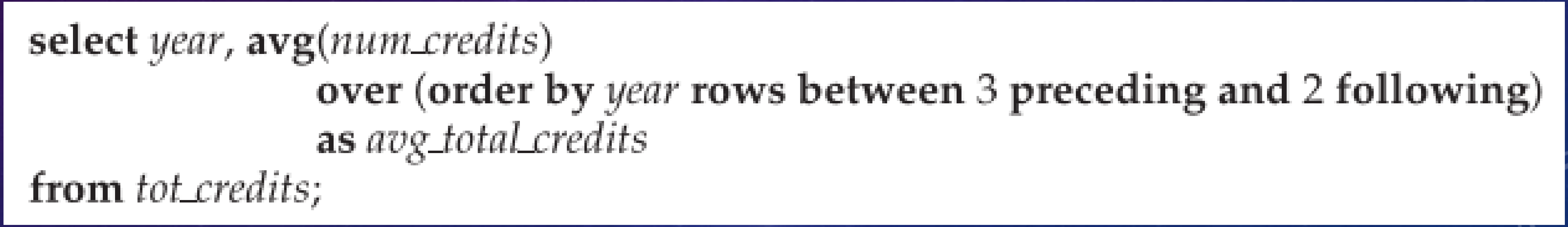
    select *date*, **sum**(*value*) **over**
        (**order by** *date* **between rows** 1 **preceding and** 1 **following**)
    from *sales*

ADVANCED AGGREGATION FEATURES: 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

ADVANCED AGGREGATION FEATURES: WINDOWING

- following



```
select year, avg(num_credits)
                over (order by year rows between 3 preceding and 2 following)
                as avg_total_credits
from tot_credits;
```

ADVANCED AGGREGATION FEATURES: WINDOWING

- 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
```

OBJECTIVES

- Accessing SQL from a Programming Language
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

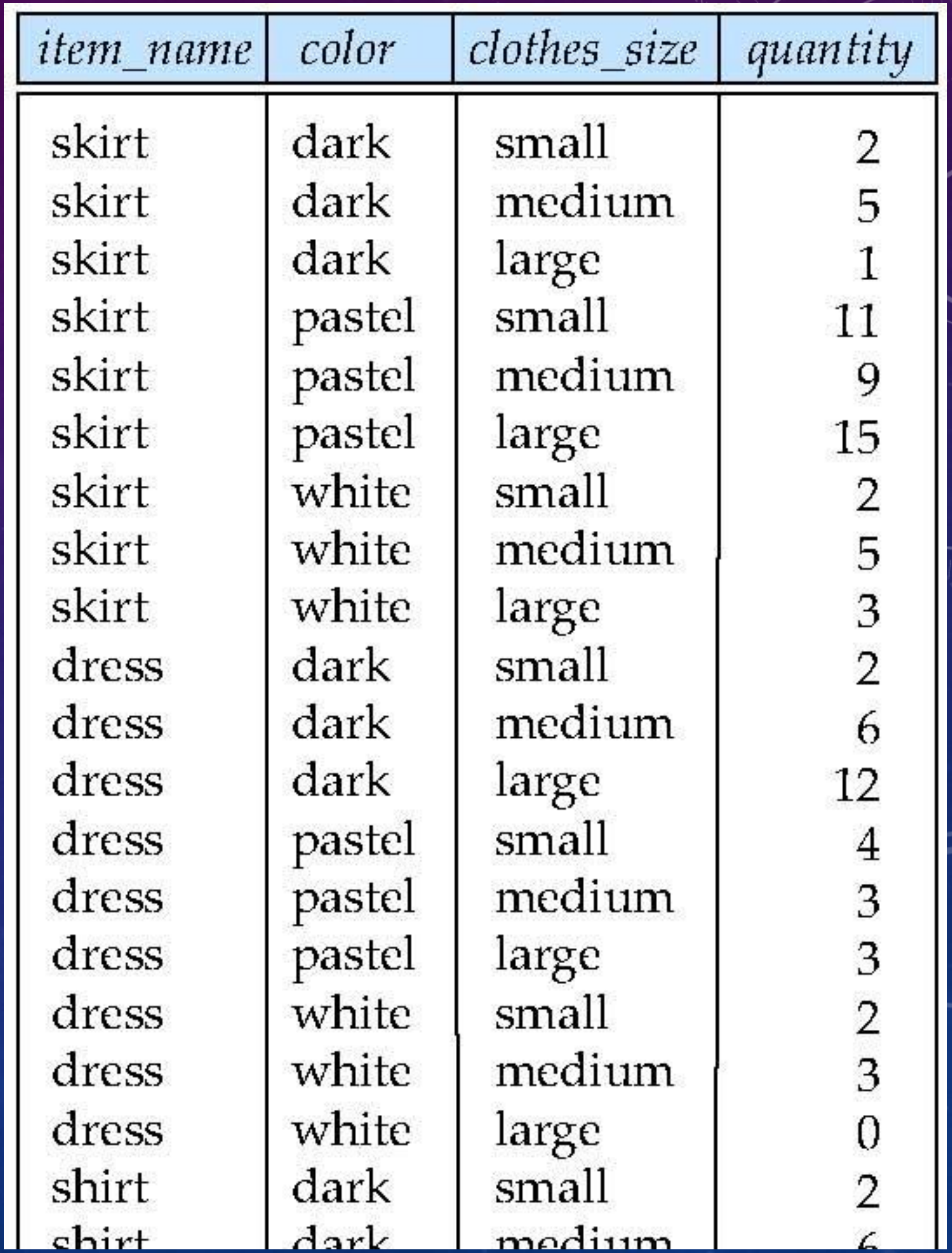
OBJECTIVES

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

OLAP

- Online Transaction Processing (OLTP)
  - transaction
- 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)

EXAMPLE: SALES RELATION

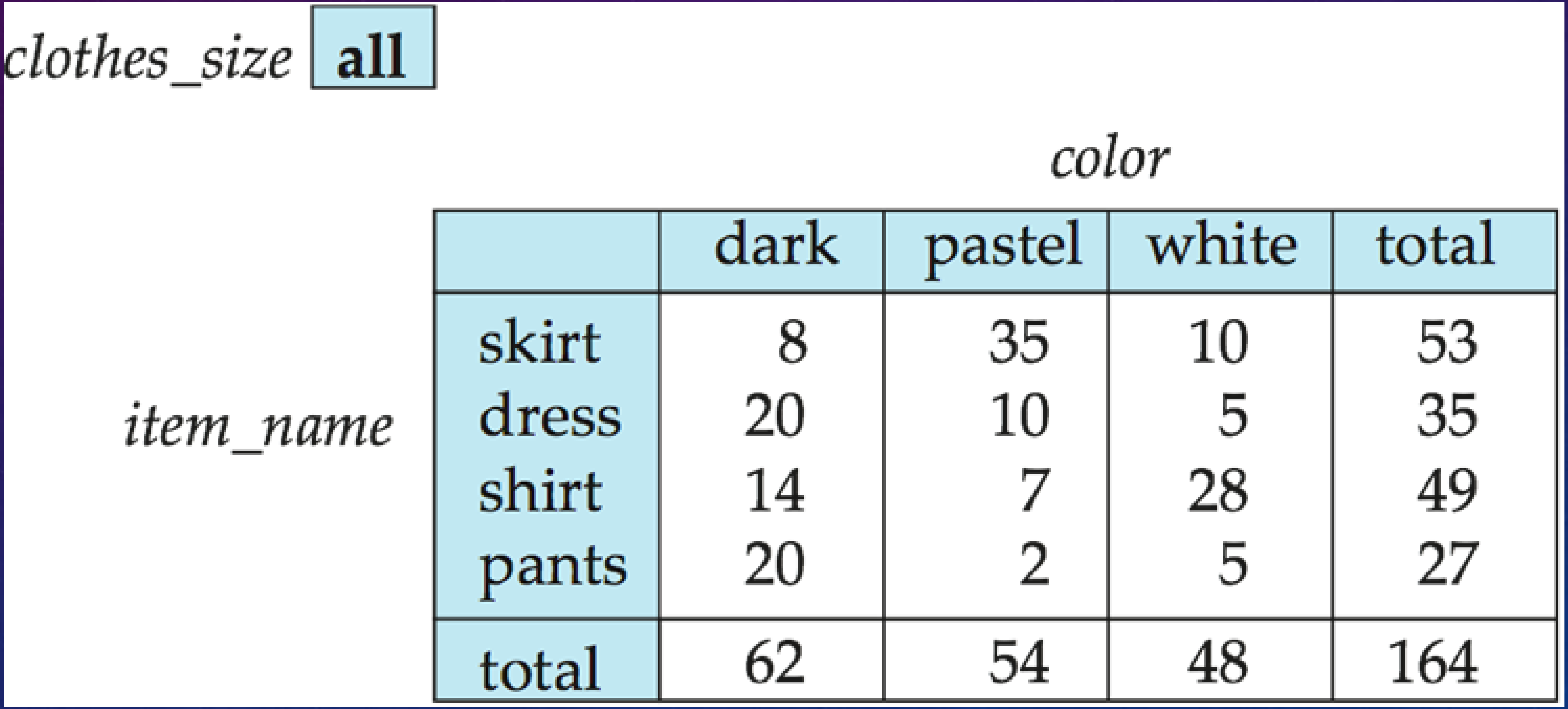


item_name	*color*	*clothes_size*	*quantity*
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel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

OLAP

- Data that can be modeled as dimension attributes and measure attributes are called **multidimensional多维 data**.
  - **Measure attributes 度量属性**
    - measure some value
    - can be aggregated upon
    - e.g., the attribute *quantity* of the *sales* relation
  - **Dimension attributes维属性**
    - define the dimensions on which measure attributes (or aggregates thereof) are viewed
    - e.g., attributes *item_name, color,* and *size* of the *sales* relation

CROSS TABULATION交叉表 OF SALES BY ITEM_NAME AND COLOR



clothes_size **all**

color

item_name

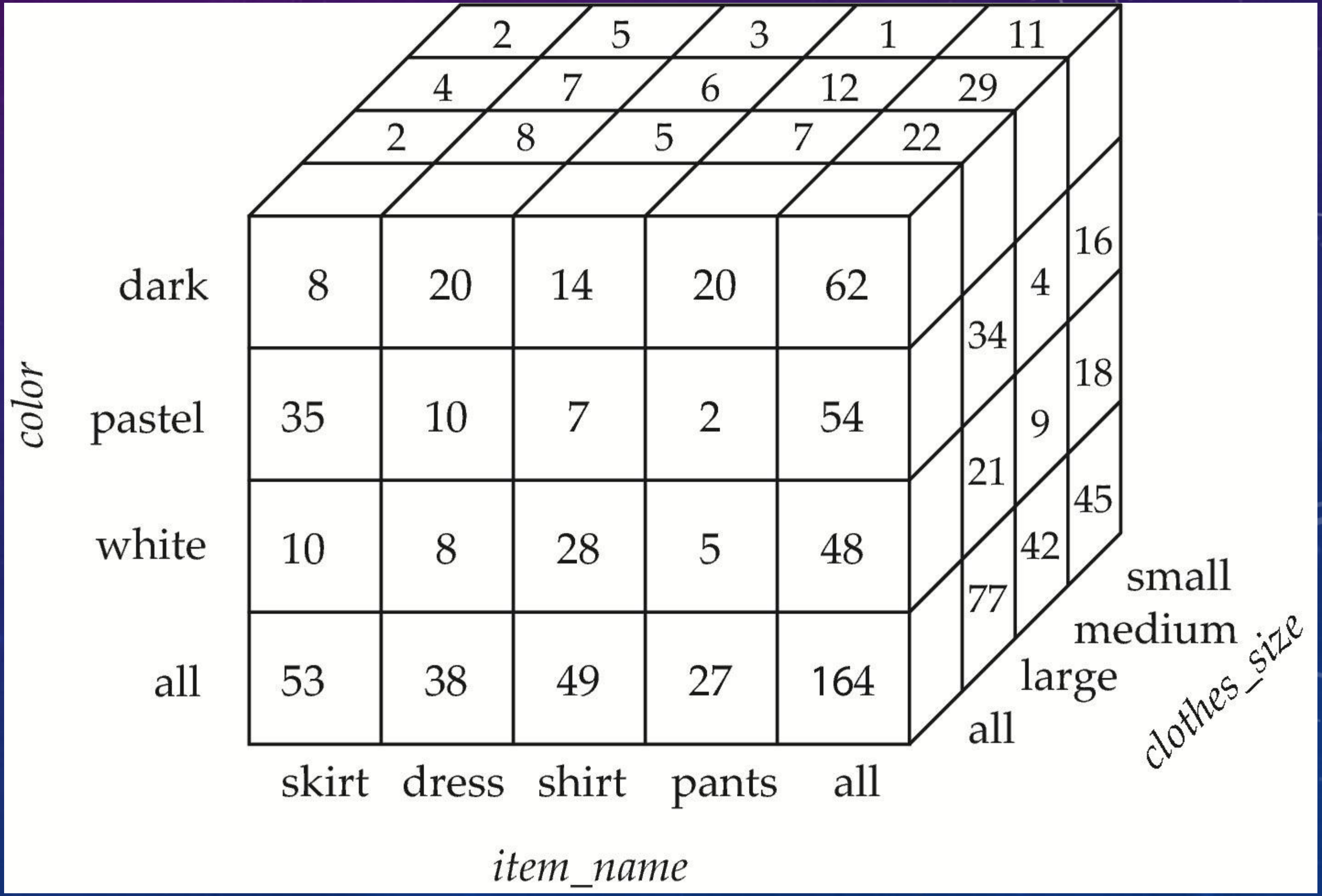
	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

CROSS TABULATION交叉表 OF SALES BY ITEM_NAME AND COLOR

- The table above is an example of a **cross-tabulation** (**cross-tab**)交叉表, also referred to as a **pivot-table**转轴表.
  - 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数据立方体

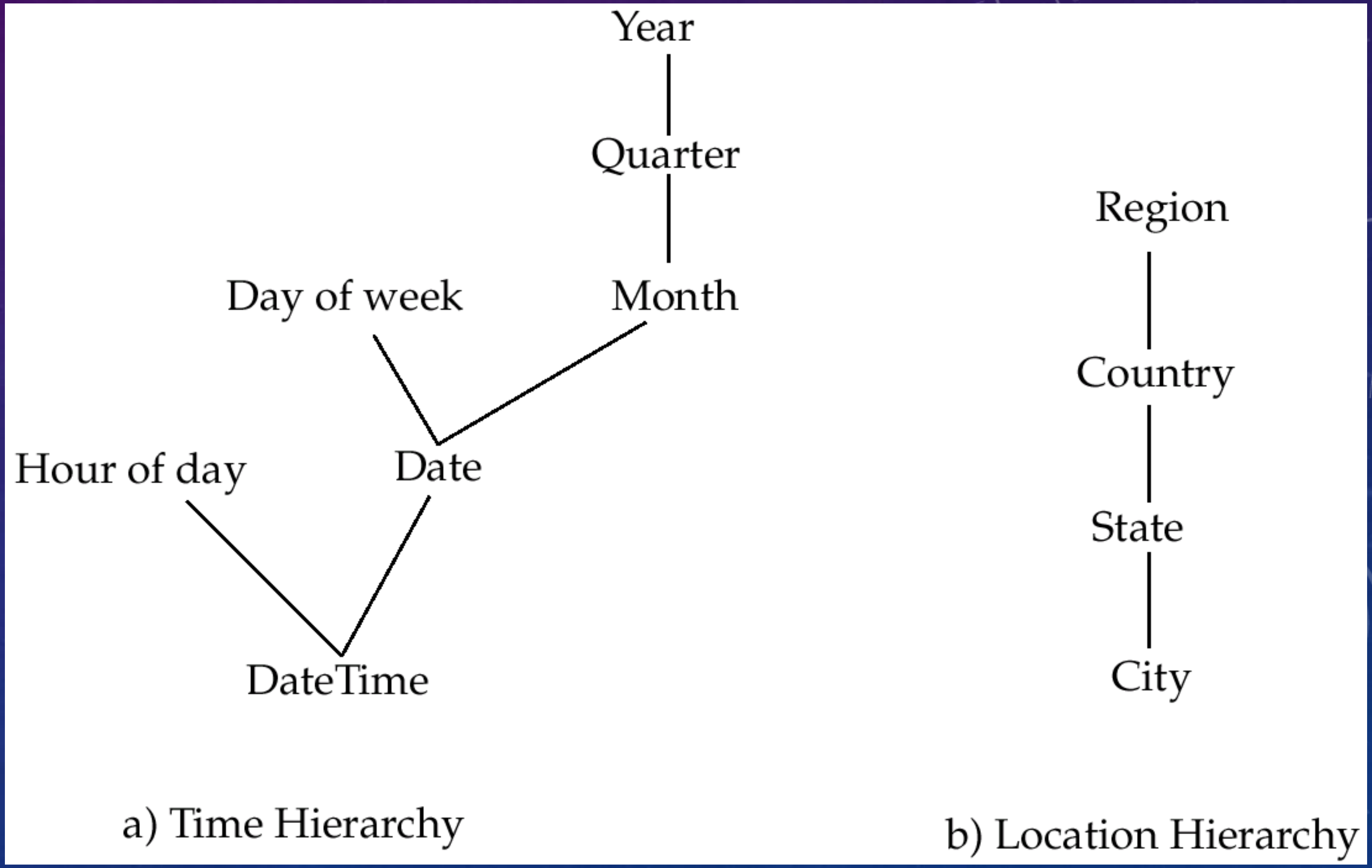
- A **data cube** is a multidimensional generalization of a cross-tab
- Can have ***n*** dimensions（n维）; we show 3 below
- Cross-tabs can be used as views on a data cube



color	skirt	dress	shirt	pants	all
dark	8	20	14	20	62
pastel	35	10	7	2	54
white	10	8	28	5	48
all	53	38	49	27	164

HIERARCHIES ON DIMENSIONS

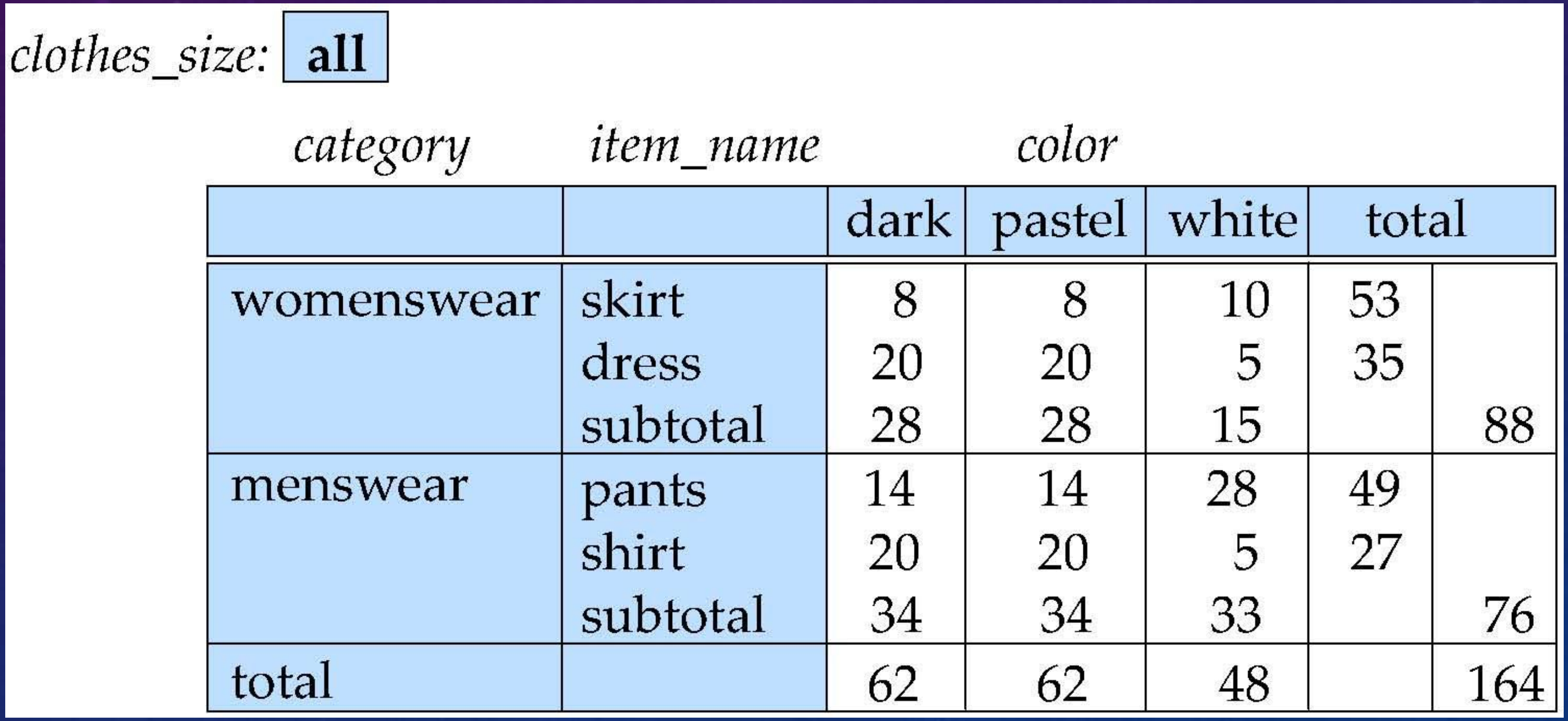
- **Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail
  - E.g., the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year



©LXD

CROSS TABULATION WITH HIERARCHY

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



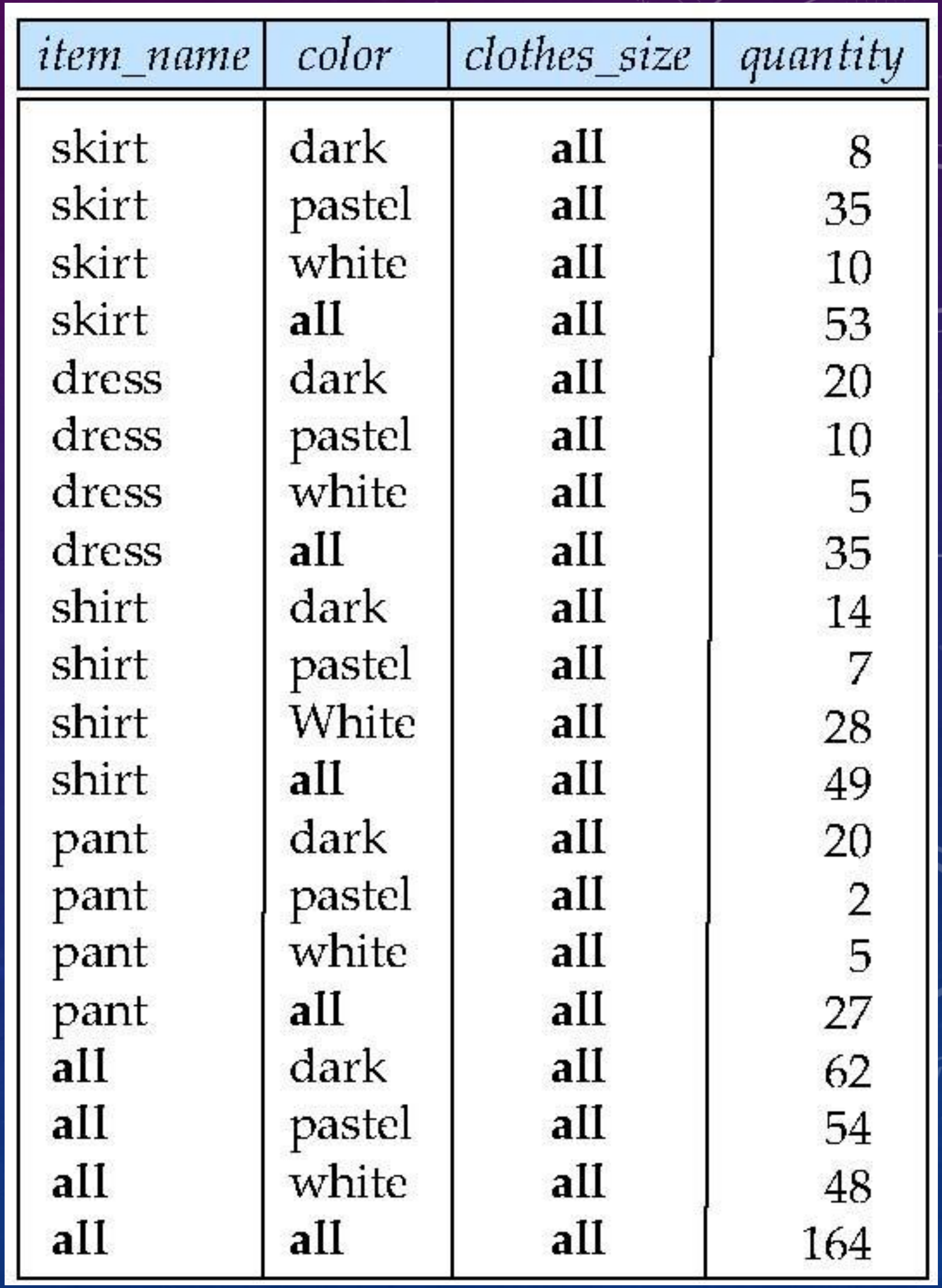
clothes_size: **all**

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

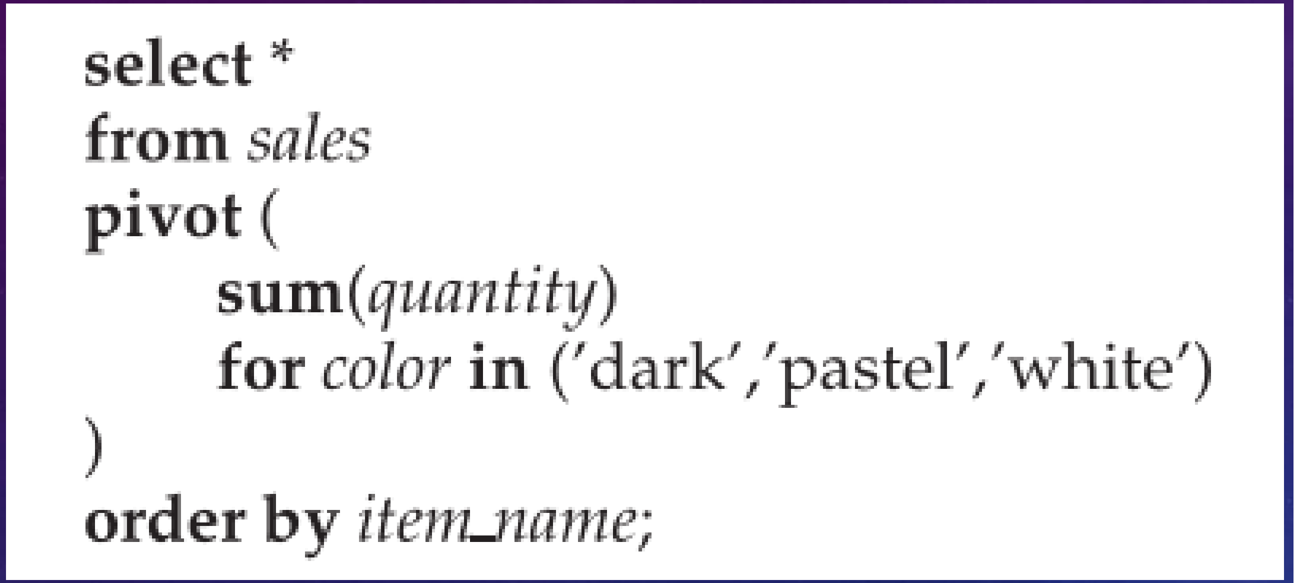
- Cross-tabs can be represented as relations
  - We use the value **all** is used to represent aggregates.
  - The SQL standard actually uses null values in place of **all** despite confusion with regular null values.

©LXD

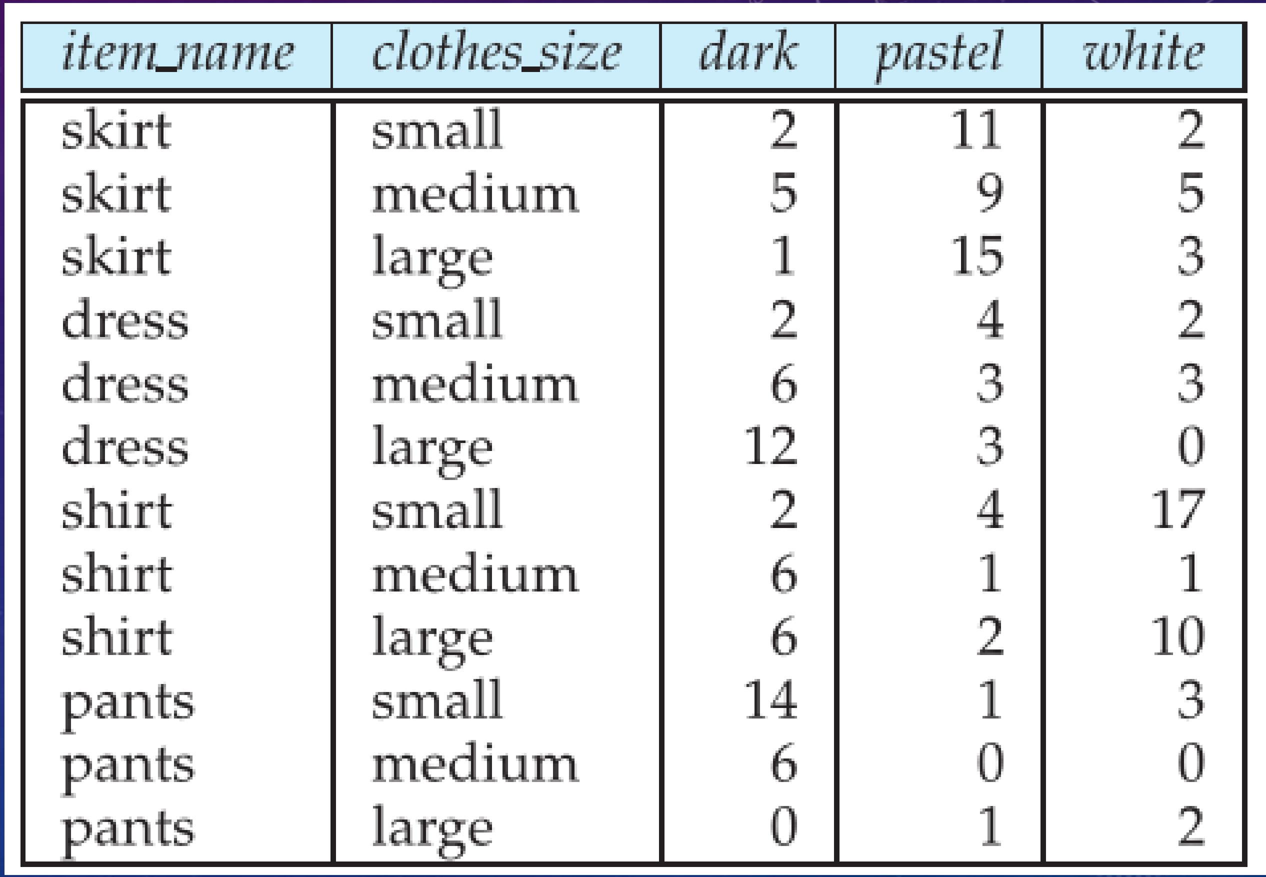


item_name	*color*	*clothes_size*	*quantity*
skirt	dark	all	8
skirt	pastel	all	35
skirt	white	all	10
skirt	all	all	53
dress	dark	all	20
dress	pastel	all	10
dress	white	all	5
dress	all	all	35
shirt	dark	all	14
shirt	pastel	all	7
shirt	White	all	28
shirt	all	all	49
pant	dark	all	20
pant	pastel	all	2
pant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

PIVOT转轴



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

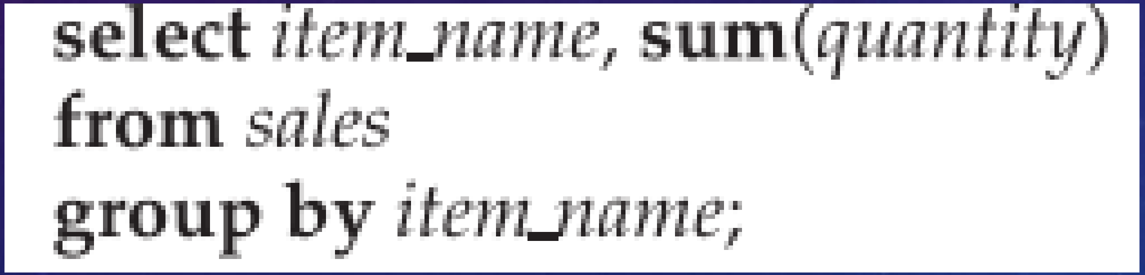


item_name	*clothes_size*	*dark*	*pastel*	*white*
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
pants	small	14	1	3
pants	medium	6	0	0
pants	large	0	1	2

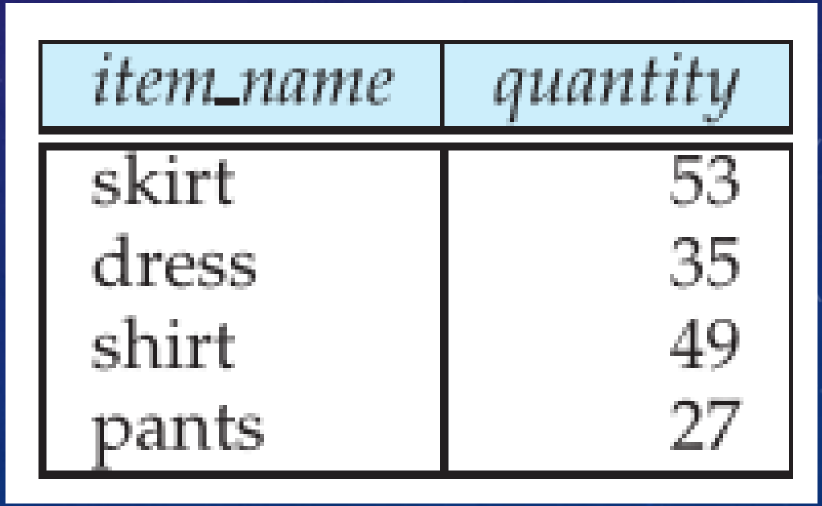
EXTENDED AGGREGATION TO SUPPORT OLAP

- 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*)



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



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

EXTENDED AGGREGATION TO SUPPORT OLAP（CONT.，）

- The **cube** operation computes union of **group by**’s on every subset of the specified attributes
- Example relation for this section:

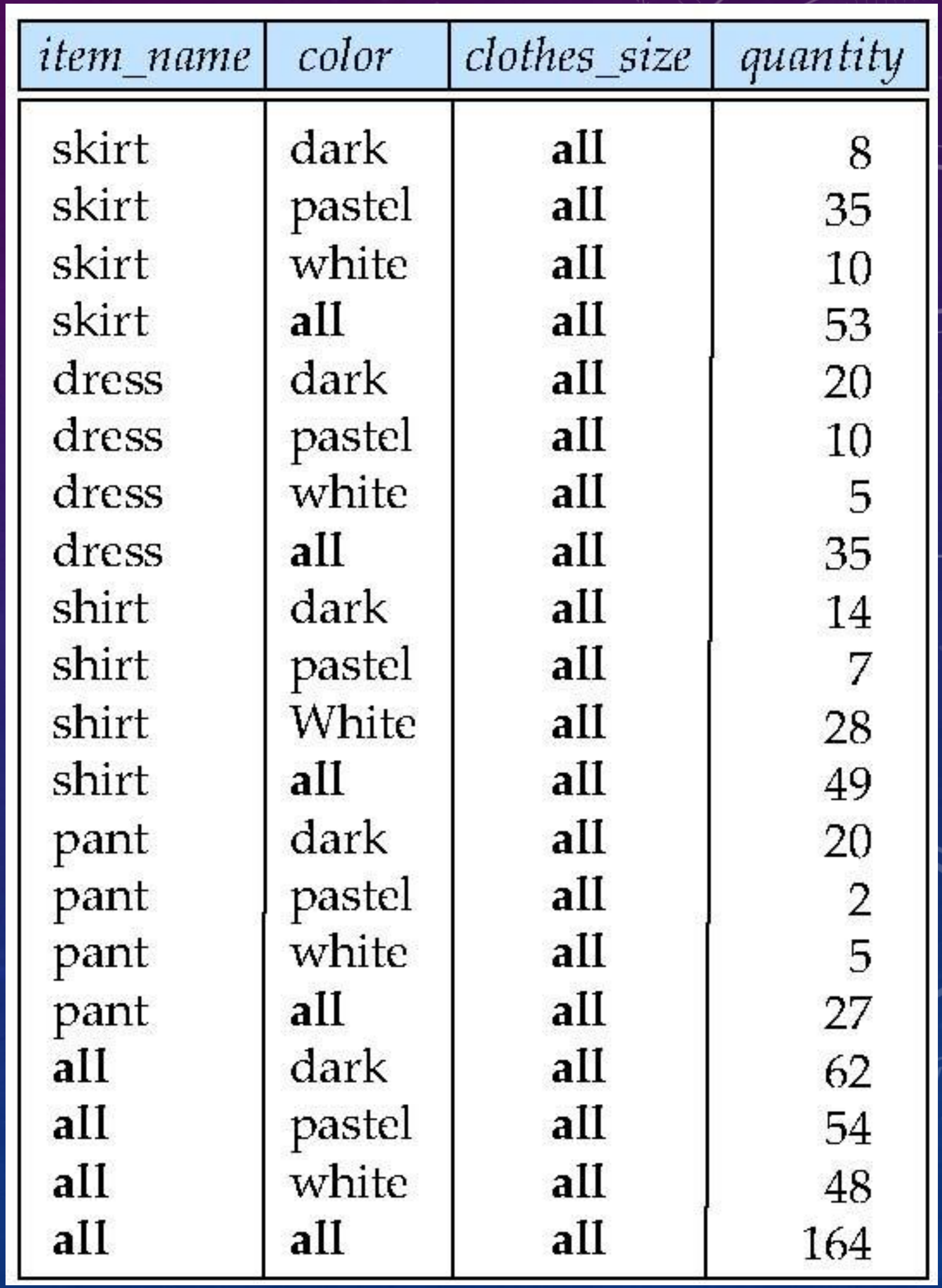
  sales(*item_name, color, clothes_size, quantity*)

- E.g. consider the query

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

EXTENDED AGGREGATION TO (CONT.,)

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



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 (CONT.,)

- 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: GROUPING

- 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*)

ONLINE ANALYTICAL PROCESSING OPERATIONS: GROUPING (CONT.,)

- 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: GROUPING （CONT.,）

- Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**
  - decode (value, match-1, replacement-1, match-2, replacement-2, ..., match-N, replacement-N, default-replacement);
  - E.g., replace *item_name* in first query by
    decode(**grouping**(item_*name*), 1, 'all', *item_name*)

ONLINE ANALYTICAL PROCESSING OPERATIONS: ROLLUP上卷

- The **rollup** construct generates union on every prefix of specified list of attributes 由细粒度到粗粒度
- 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*), ( ) }

ONLINE ANALYTICAL PROCESSING OPERATIONS: ROLLUP (CONT.,)

- 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.*

ONLINE ANALYTICAL PROCESSING OPERATIONS

- 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转轴:** changing the dimensions used in a cross-tab is called
- **Slicing分片:** creating a cross-tab for fixed values only
    - Sometimes called **dicing切块**, particularly when values for multiple dimensions are fixed.
- **Rollup上卷:** moving from finer-granularity data to a coarser granularity
- **Drill down下钻:** 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.

- 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

SUMMARY

- Accessing SQL from a Programming Language
- Temporary Table
- Functions and Procedures
- Triggers
- Recursive Queries
- Advanced Aggregation Features
- OLAP

Q&A?

THANKS！

leexudong@nankai.edu.cn