Database Systems

Lecture 6:
Advanced SQL

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Outline

- **Accessing SQL From a Programming Language**
- Functions
- Triggers
- 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

- Possible approaches:
  - Dynamic SQL
    - JDBC (Java Database Connectivity) works with Java
    - ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic.
      - Other API's such as ADO.NET sit on top of ODBC
  - 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 Example

```
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 Example

```
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);
                }
```

JDBC Example

```
            }
            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);
        }
    }
```

Database Connection

- Each database product that supports JDBC provides a JDBC driver that must be dynamically loaded in order to access the database from Java.
  - This is done by invoking *Class.forName* with one argument specifying a concrete class implementing the java.sql.Driver interface
- Connecting to the Database: A connection is opened using the getConnection method of the DriverManager class (within java.sql) using 3 parameters:
  - a string that specifies the URL, or machine name, where the server runs
  - a database user identifier, which is a string
  - a password, which is also a string.
    - Note: the need to specify a password within the JDBC code presents a security risk if an unauthorized person accesses your Java code.

Shipping SQL Statements

- Methods for executing a statement:
  - *executeQuery*
    - When the SQL statement is a query
    - It returns a result set
  - *executeUpdate*
    - When the SQL statement is nonquery (DDL or DML)
      - Update
      - Insert
      - Delete
      - Create table
      - …
    - It returns an integer giving the number of tuples inserted, updated, or deleted.
    - For DDL statements, the return value is zero.

Retrieving the Results of a Query

- Retrieving the set of tuples in the result into a ResultSet object
- Fetching the results one tuple at a time
- Using the *next* method on the result set to test whether there remains at least one unfetched tuple in the result set and if so, fetches it.
- Attributes from the fetched tuple are retrieved using various methods whose names begin with *get*
  - *getString:* can retrieve any of the basic SQL data types
  - *getFloat*
- Possible argument to the *get* methods
  - The attribute name specified as a string
  - An integer indicating the position of the desired attribute within the tuple

Database Connection

- The statement and connection are both closed at the end of the Java program.
- It is important to close the connection because there is a limit imposed on the number of connections to the database
- Unclosed connections may cause that limit to be exceeded.
- If this happens, the application cannot open any more connections to the database.

Prepared Statements

- Creating a prepared statement in which some values are replaced by "?"
- Specifying that actual values will be provided later
- Compiling the query by the database system when it is prepared
- Reusing the previously compiled form of the query and apply the new values whenever the query is executed
  - (with new values to replace the "?"s),

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();
```

Figure 5.2 Prepared statements in JDBC code.

- insert into instructor values (“88877”, “Perry”, “Finance”, 125000)
- insert into instructor values (“88878”, “Perry”, “Finance”, 125000)

Prepared Statements

- Prepared statements allow for more efficient execution
  - Where the same query can be compiled once and then run multiple times with different parameter values
  - Whenever a user-entered value is used, even if the query is to be run only once

Metadata Features

- Capturing metadata about
  - Database
  - ResultSet (relations)

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

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.

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**
- The SQL structures permitted in the host language comprise *embedded* SQL
- An embedded SQL program must be processed by a special preprocessor prior to compilation.
- The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow runtime execution of the database accesses.
- Then, the resulting program is compiled by the host-language compiler.

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 { …. };

Database Connection

- 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

- 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)*

- 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;
```

SQL Query

- To write an embedded SQL query, we use the following statement:

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

- The variable *c* is used to identify the query

- Example:

  EXEC SQL

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

  END_EXEC

SQL Query

- The open statement is then used to evaluate the query

- 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.

SQL Query

- The fetch statement
  - Placing the values of one tuple in the query result into host language variables
  - Requiring one host-language variable for each attribute of the result relation;

    e.g., we need one variable to hold the ID value (si) and another to hold the name value (sn) which have been declared within a DECLARE section

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

Repeated calls to fetch get successive tuples in the query result

SQL Query

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL **close** *c* ;

Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)

- Example for updating 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

- Iterating through the tuples by performing **fetch** operations on the cursor
- Executing the following statement

EXEC SQL

update *instructor*
set *salary* = *salary* + 1000
where current of *c;*

Outline

- Accessing SQL From a Programming Language
- **Functions**
- Triggers
- 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 (e.g., C, Java).
  - Some database systems support **table-valued functions**, which can return a relation as a result.
- SQL:1999 also supports a rich set of imperative constructs, including
  - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999.

SQL Functions

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

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

SQL Functions

- The function *dept*_count can be used in a query that returns 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)
```

- Usage

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

Outline

- Accessing SQL From a Programming Language
- Functions
- **Triggers**
- Advanced Aggregation Features
- 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 introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

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.

Trigger Example: Using set Statement

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

Trigger Example: to Maintain Referential Integrity

```
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;

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;
```

Trigger Example: 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;
```

Disabling Triggers

- Triggers can be disabled or enabled;
  - by default they are enabled when they are created.
- Triggers can be disabled by:

  alter trigger *trigger_name* **disable**

  disable trigger *trigger_name*

- A trigger that has been disabled can be enabled again.
- A trigger can instead be dropped, which removes it permanently, by:

  drop trigger *trigger_name*

When Not To Use Triggers

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

When Not To Use Triggers

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

Outline

- Accessing SQL From a Programming Language
- Functions
- Triggers
- **Advanced Aggregation Features**
- OLAP

Ranking

- Suppose we are given a relation
  student_grades(ID, GPA)
  giving the grade-point average of each student

- Goal: finding the rank of each student.

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

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

Ranking

- Ranking is done in conjunction with an order by specification.

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

- NOTE: the extra **order by** clause is needed to get them in sorted order

Ranking

- Two possible approaches for ranking
  - Leaving gaps: (default)

    e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  - Without gaps: (using **dense_rank)**

    so next dense rank would be 2

Ranking with Partitions

- 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*;

- Ranking is done *after* applying **group by** clause/aggregation

Top *n* Items

- 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

Other Ranking Functions

- **ntile**:
  - 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;*

Other Ranking Functions

- 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*

Outline

- Accessing SQL From a Programming Language
- Functions
- Triggers
- Advanced Aggregation Features
- **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)
- Is used for **multidimensional data** (data that can be modeled as dimension attributes and measure attributes)
  - **Measure attributes**
    - measure some value
    - can be aggregated upon
    - e.g., the attribute *number* of the *sales* relation
  - **Dimension attributes**
    - define the dimensions on which measure attributes (or aggregates there of) are viewed
    - e.g., attributes *item_name, color,* and *size* of the *sales* relation

Example sales relation

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

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

clothes_size **all**

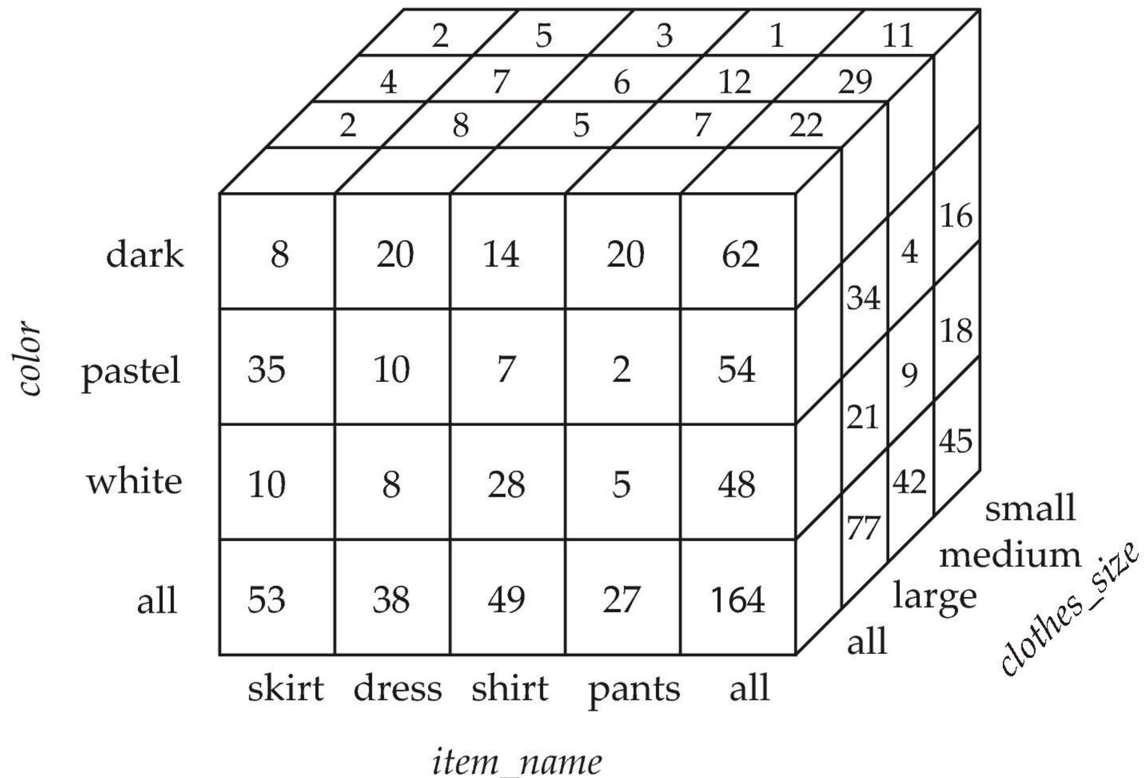
color

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

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

Data Cube

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



Online Analytical Processing Operations

- **Pivoting:** changing the dimensions used in a cross-tab
  - Each cross-tab is a two-dimensional view on a multidimensional data cube.
  - With an OLAP system, a data analyst can look at different cross-tabs on the same data by interactively selecting the attributes in the cross-tab.
  - Examples:
    - selecting a cross-tab on item name and clothes size
    - selecting a cross-tab on color and clothes size

Online Analytical Processing Operations

- **Slicing:** creating a cross-tab for fixed values only
  - OLAP systems allow an analyst to see a cross-tab on item name and color for a fixed value of clothes size,
  - Example:
    - large, instead of the sum across all sizes.
  - This operation is referred to as slicing, since it can be thought of as viewing a slice of the data cube.
  - The operation is sometimes called **dicing**.

Cross Tabulation With Hierarchy

- The following table can be achieved using natural join with another table specifying item_names and category

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

Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
- Can drill down or roll up on a hierarchy
  - roll up: moving from finer granularity to coarser-granularity (by the mean of aggregation)
  - drill down: moving from coarser-granularity to finer granularity (must be generated from original data)

Relational Representation of Cross-tabs

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

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

Extended Aggregation to Support OLAP

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

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

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

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

where ( ) denotes an empty **group by** list.

- For each grouping, the result contains the null value for attributes not present in the grouping.

Extended Aggregation

- The **rollup** construct generates union on every prefix of specified list of attributes

- E.g.,

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

Generates union of four groupings:

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

Extended Aggregation

- Rollup can be used to generate aggregates at multiple levels of a hierarchy.

- E.g., suppose table *itemcategory*(*item_name, category*) gives the category of each item. Then

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

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

Extended Aggregation

- Multiple rollups or 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*), ( ) }

OLAP Implementation

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

OLAP Implementation

- 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**

- Several optimizations available for computing multiple aggregates
  - 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*)
      - is cheaper than computing it from scratch

Questions?