Advanced SQL

CS3043 - Database Systems

Department of Computer Science and Engineering
University of Moratuwa

Overview

- JDBC
- Prepared Statements and SQL injection
- ODBC
- Embedded SQL
- Procedural constructs in SQL
- Triggers
- Advanced aggregation features
- Authorization

JDBC and ODBC

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

JDBC

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

JDBC example

```
import java.sql.*;

….

static final String DB_URL = "jdbc:mysql://localhost/university";
static String USER;
static String PASSWORD;
static final String QUERY = "SELECT ID,name FROM student";

….

try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASSWORD);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(QUERY);) {
    // Extract data from result set
    while (rs.next()) {
        // Retrieve by column name
        System.out.println("ID: " + rs.getInt("id"));
        System.out.println("Name: " + rs.getString("name"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

JDBC examples (cont'd)

- Update to database

```
try {
        statement.executeUpdate("INSERT INTO instructor VALUES
                                                ('77987','Kim','Physics',98000)");
} catch (SQLException e) {
   log.error("Unable to access the database server: ", e);
}
```

- Execute query and fetch results

```
ResultSet resultSet = statement.executeQuery("SELECT dept_name, AVG(salary)
                                                FROM instructor GROUP BY dept_name"
                                                );
while (resultSet.next()) {
       System.out.println(resultSet.getString("dept_name") + " : " +
                                                resultSet.getFloat(2));
}
```

JDBC Code

- Getting result fields
  - `resultSet.getString("dept_name")`
  - `resultSet.getString(1)`

equivalent if `dept_name` is the first attribute of the SELECT result

- Dealing with NULL values
  - If a result set is empty after executing a statement it should be handled properly.
  - As a best practice, always prepare for potential null values after executing a query, since you are not aware of the result set until you execute the query.

Prepared Statement

```
PreparedStatement preparedStatement = connection.prepareStatement
                                        ("INSERT INTO instructor
                                                VALUES(?,?,?,?)");
preparedStatement.setString( 1, "88877");
preparedStatement.setString( 2, "Perry");
preparedStatement.setString( 3, "Finance");
preparedStatement.setInt( 4, 125000);
preparedStatement.executeUpdate();
```

- For queries, use `preparedStatement.executeQuery()` which returns a `ResultSet`.
- Always use prepared statements when taking an input from the user and adding it to a query

Prepared Statement

```
PreparedStatement preparedStatement = connection.prepareStatement
                                        ("INSERT INTO instructor
                                            VALUES(?,?,?,?)");
preparedStatement.setString( 1, "88877");
preparedStatement.setString( 2, "Perry");
preparedStatement.setString( 3, "Finance");
preparedStatement.setInt( 4, 125000);
preparedStatement.executeUpdate();
```

- For queries, use `preparedStatement.executeQuery()` which returns a `ResultSet`.
- Always use prepared statements when taking an input from the user and adding it to a query
    - **NEVER create a query by concatenating strings taken as user inputs**

```
"INSERT INTO instructor VALUES(' " + ID + " ', ' " + name + " ', " + " ' +
dept name + " ', " ' balance + ")"
```

    - **What if the name is “D’Souza”?**

SQL Injection

- Suppose a query is constructed using
  - `"select * from instructor where name = '" + name + "'"`
- Suppose the user enters the following, instead of entering the name
  - X' or 'Y' = 'Y
- Then the resulting query becomes:
  - `"select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"`
  - which is

    ```
    select * from instructor where name = 'X' or 'Y' = 'Y'
    ```
- Prepared Statements internally sanitize the inputs.

  ```
  select * from instructor where name = 'X\' or \'Y\' = \'Y'
  ```

- Always use Prepared Statements if the parameters for a query are taken as user inputs.

Transaction control in JDBC

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

Other JDBC features

- Handling large object types
  - `getBlob()` and `getClob()` can be used to return objects of type Blob or Clob, respectively.
  - associate an open stream with Java Blob or Clob object to update large objects

```
blob.setBlob(int parameterIndex, InputStream inputStream)
```

- Metadate features
  - After executing a query to get the result set, you can access the metadata of the result set by,

```
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 Standard

- Standard for an application program to communicate with a database server.
- Application program interface (API) to
  - open a connection with a database,
  - send queries and updates,
  - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC
- Was defined originally for Basic and C, versions available for many languages.
- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.

ODBC example

```
int ODBCexample()
{
      RETCODE error;
      HENV env; /* environment */
      HDBC conn; /* database connection */
      SQLAllocEnv(&env);
      SQLAllocConnect(env, &conn);
      SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
                           "avipasswd", SQL_NTS);

      { …. Do actual work … }

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

Read more about ODBC from Chapter 4 of the recommended text.

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol.
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- An embedded SQL program must be pre-processed by a special preprocessor prior to compilation.
  - The preprocessor replaces embedded SQL commands with host language declarations that allow run-time execution.
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

```
EXEC SQL <embedded SQL statement > END_EXEC
```

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

SQLJ

- JDBC is overly dynamic - errors cannot be caught by the Java compiler
- SQLJ: embedded SQL in Java
- How does embedding help to detect errors early?

```
#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();
```

Procedural Constructs in SQL

Procedural Extensions and Stored Procedures

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

SQL Functions

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

SQL Functions

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

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

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

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

Table Functions

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

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

- Usage

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

SQL Procedures

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

```
create procedure dept_count_proc (in dept_name varchar(20),
                                  out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dept_count_proc.dept_name
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

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

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

Procedural Constructs

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

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

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

Procedural Constructs

- For loop
  - Permits iteration over all results of a query

    Example:

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

External Language Functions/Procedures

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

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


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

External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - more efficient for many operations, and more expressive power.
- Drawbacks
  - Code to implement function may need to be loaded into the 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 provide security at the cost of potentially worse performance.
    - Use **sandbox** techniques
    - run external language functions/procedures in a separate process, with no access to the memory of the database process
  - Direct execution in the database system's space is used when efficiency is more important than security.

Triggers

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

Trigger example

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

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

Trigger example (cont'd)

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

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes

  E.g., after update of *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as :** for deletes and updates
  - **referencing new row as :** for inserts and updates
- Triggers can be activated before an event which can serve as extra constraints.

  E.g. convert blank grades to null.

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

Statement Level Triggers

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

When Not To Use Triggers

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

When Not To Use Triggers

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

Advanced Aggregation Features

Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation; *student_grades(ID, GPA)* giving the grade-point average of each student. Find the rank of each student.

  select *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s rank _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
- Supported from MySQL 8.0.2. Syntax may differ, please check the manual.

Windowing

- Used to smooth out random variations.

  E.g., **moving average**: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”
- **Window specification** in SQL:
  - Given relation *sales(date value) , 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**
  - **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
- Supported from MySQL 8.0.2.

Authorization

Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

```
grant <privilege list>
on <relation name or view name> to <user list>
```

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the required privileges on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view

  Example: grant users *U*1, *U*2, and *U*3 **select** authorization on the *instructor* relation:

  ```
  grant select on instructor to U1, U2, U3
  ```

- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>
  on <relation name or view name> **from** <user list>

- Example:

  revoke select on *branch* **from** *U1 U2 U3*

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- **create role** instructor;
- **grant** *instructor* **to Amit;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor;*
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant;*
  - **grant** *teaching_assistant* **to** *instructor;*
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean;*
  - **grant** *instructor* **to** *dean;*
  - **grant** *dean* **to** Satoshi;

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option;**
  - **revoke select on** *department* **from** Amit, Satoshi **cascade;**
  - **revoke select on** *department* **from** Amit, Satoshi **restrict;**
- Read Section 4.6 for more details we have omitted here.

Thank you!

OLAP
Online Analytical Processing

Out of the scope of CS3043

OLAP - Online Analytical Processing

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

Example - Cross Tabulation of sales by item_name and color

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
...	...	...	...
...	...	...	...

Sales relation

clothes_size **all**

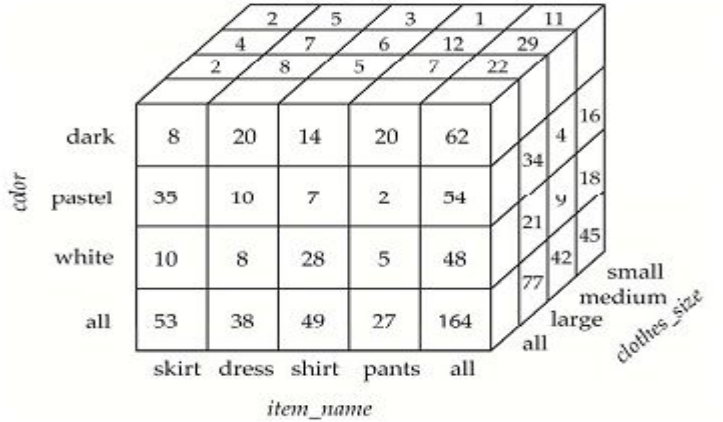
item_name \ *color*	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

The table above is an example of a **cross-tabulation** (**cross-tab**), also referred to as a **pivot-table**.

- Values for one of the dimension attributes form the row headers
- Values for another dimension attribute form the column headers
- Other dimension attributes are listed on top
- Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

Data Cube

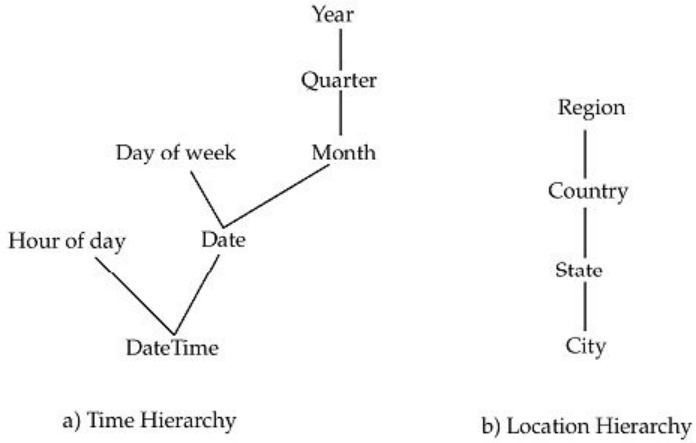
- A **data cube** is a multidimensional generalization of a cross-tab
- Can have *n* dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube



Hierarchies on Dimensions

- **Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail

    E.g., the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year



a) Time Hierarchy

b) Location Hierarchy

Cross Tabulation With Hierarchy

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

clothes_size: **all**

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

Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
  - 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

Online Analytical Processing Operations

- **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:** 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

Reading material

Extended Aggregation to Support OLAP

- The **cube** operation computes union of **group by**'s on every subset of the specified attributes

  Example relation for this section

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

- E.g. consider the query

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

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

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

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

Online Analytical Processing Operations

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

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

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

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

Online Analytical Processing Operations

- Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**

  E.g., replace *item_name* in first query by

  decode(**grouping**(item_*name*), 1, ‘all’, *item_name*)

Extended Aggregation (Cont.)

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

  E.g.,

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

- Generates union of four groupings:

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

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

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

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

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

Extended Aggregation (Cont.)

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

```
select item name, color, size _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.

Procedural Constructs - Exception Handling

- 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

Quiz

	popcorn	oil amt	batch	yield
1	plain	little	large	8.2
2	gourmet	little	large	8.6
3	plain	lots	large	10.4
4	gourmet	lots	large	9.2
5	plain	little	small	9.9
6	gourmet	little	small	12.1
7	plain	lots	small	10.6
8	gourmet	lots	small	18.0
9	plain	little	large	8.8
10	gourmet	little	large	8.2
11	plain	lots	large	8.8
12	gourmet	lots	large	9.8
13	plain	little	small	10.1
14	gourmet	little	small	15.9
15	plain	lots	small	7.4
16	gourmet	lots	small	16.0

Create the data cube for the following table

Explain how you find the average yield for plain popcorn with little oil amount.