Chapter 5: Advanced SQL
Outline

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

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

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

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

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a “statement” object
  - Execute queries using the statement object to send queries and fetch results
  - Exception mechanism to handle errors
public static void JDBCexample(String dbid, String userid, String passwd) {
    try (Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
         Statement stmt = conn.createStatement();)
    {
        \(\ldots\) Do Actual Work \(\ldots\)
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}

NOTE: Above syntax works with Java 7, and JDBC 4 onwards. Resources opened in “try (....)” syntax (“try with resources”) are automatically closed at the end of the try block
JDBC Code for Older Versions of Java/JDBC

```java
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        … Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : "+ sqle);
    }
}

NOTE: Class.forName is not required from JDBC 4 onwards. The try with
resources syntax in prev slide is preferred for Java 7 onwards.
```
JDBC Code (Cont.)

- Update to database

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

- Execute query and fetch and print results

```java
ResultSet rset = stmt.executeQuery("select dept_name, avg(salary) from instructor group by dept_name");
while (rset.next()) {
    System.out.println(rset.getString("dept_name") + " " + rset.getFloat(2));
}
```
JDBC SUBSECTIONS

- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- Callable Statements
- Metadata Features
- Other Features
- Database Access from Python
JDBC Code Details

- Getting result fields:
  - \( \text{rs.getString("dept_name")} \) and \( \text{rs.getString(1)} \) equivalent if \text{dept_name} is the first argument of select result.

- Dealing with Null values

  ```java
  int a = rs.getInt("a");
  if (rs.wasNull()) System.out.println("Got null value");
  ```
Premade Statement

- `PreparedStatement pStmt = conn.prepareStatement(
  "insert into instructor values(?,?,?,?)");`
- `pStmt.setString(1, "88877");`
- `pStmt.setString(2, "Perry");`
- `pStmt.setString(3, "Finance");`
- `pStmt.setInt(4, 125000);`
- `pStmt.executeUpdate();`
- `pStmt.setString(1, "88878");`
- `pStmt.executeUpdate();`

- **WARNING:** always use prepared statements when taking an input from the user and adding it to a query
  - NEVER create a query by concatenating strings
  - "insert into instructor values(\" + ID + ", \" + name + ", \" + dept name + ", \" balance + ")""
  - What if name is “D'Souza”?
SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = "" + name + ""
- Suppose the user, instead of entering a name, enters:
  - 'X' or 'Y' = 'Y'
- Then the resulting statement becomes:
  - "select * from instructor where name = "" + "X' or 'Y' = 'Y" + ""
  - Which is:
    - select * from instructor where name = 'X' or 'Y' = 'Y'
  - User could have even used
    - 'X'; update instructor set salary = salary + 10000; --
- Prepared statement internally uses:
  - "select * from instructor where name = 'X' or 'Y' = 'Y'
  - Always use prepared statements, with user inputs as parameters
Metadata Features

- ResultSet metadata
- E.g. after executing query to get a ResultSet rs:
  - ResultSetMetaData rsmd = rs.getMetaData();
    for(int i = 1; i <= rsmd.getColumnCount(); i++) {
      System.out.println(rsmd.getColumnName(i));
      System.out.println(rsmd.getColumnTypeName(i));
    }
- How is this useful?
Metadata (Cont)

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();
  // 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
  // The value null indicates all Catalogs/Schemas.
  // The value "" indicates current catalog/schema
  // The value "%" has the same meaning as SQL like clause
  ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
  while( rs.next()) {
    System.out.println(rs.getString("COLUMN_NAME"),
                       rs.getString("TYPE_NAME");
  }
- And where is this useful?
Metadata (Cont)

- Database metadata
  
  ```java
  DatabaseMetaData dbmd = conn.getMetaData();
  // Arguments to getTables: Catalog, Schema-pattern, Table-pattern, and Table-Type
  // Returns: One row for each table; row has a number of attributes such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
  // The value null indicates all Catalogs/Schemas.
  // The value "" indicates current catalog/schema
  // The value "%" has the same meaning as SQL like clause
  // The last attribute is an array of types of tables to return.
  // TABLE means only regular tables
  ResultSet rs = dbmd.getTables ("", "", "%", new String[] {"TABLES"});
  while( rs.next()) {
    System.out.println(rs.getString("TABLE_NAME"));
  }
  ```

- And where is this useful?
Finding Primary Keys

- DatabaseMetaData dmd = connection.getMetaData();

  // Arguments below are: Catalog, Schema, and Table
  // The value “” for Catalog/Schema indicates current catalog/schema
  // The value null indicates all catalogs/schemas
  ResultSet rs = dmd.getPrimaryKeys("", "", tableName);

  while(rs.next()){
    // KEY_SEQ indicates the position of the attribute in
    // the primary key, which is required if a primary key has multiple
    // attributes
    System.out.println(rs.getString("KEY_SEQ"),
                        rs.getString("COLUMN_NAME");
  }
Transaction Control in JDBC

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

- Calling functions and procedures
  - CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");
  - CallableStatement cStmt2 = conn.prepareCall("{call some procedure(? , ?)}");

- Handling large object types
  - getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively.
  - get data from these objects by getBytes()
  - associate an open stream with Java Blob or Clob object to update large objects
    - blob.setBlob(int parameterIndex, InputStream inputStream).
JDBC Resources

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

- JDBC is overly dynamic, errors cannot be caught by compiler
- SQLJ: embedded SQL in Java
  
  ```java
  #sql iterator deptInfoIter ( String dept name, int avgSal);
  deptInfoIter iter = null;
  #sql iter = { select dept_name, avg(salary) from instructor
  group by dept name };
  while (iter.next()) {
      String deptName = iter.dept_name();
      int avgSal = iter.avgSal();
      System.out.println(deptName + " " + avgSal);
  }
  iter.close();
  ```
ODBC
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
Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1.
- A language to which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language comprise embedded SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/1.
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor
  
  ```
  EXEC SQL <embedded SQL statement >;
  ```

  Note: this varies by language:
  - In some languages, like COBOL, the semicolon is replaced with END-EXEC
  - In Java embedding uses `# SQL { .... };`
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`)

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

```
EXEC-SQL BEGIN DECLARE SECTION

int credit-amount;

EXEC-SQL END DECLARE SECTION;
```
Embedded SQL (Cont.)

- To write an embedded SQL query, we use the `declare c cursor for <SQL query>` statement. The variable `c` is used to identify the query.

- Example:
  - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable `credit_amount` in the host language.
  - Specify the query in SQL as follows:
    ```sql
    EXEC SQL
    declare c cursor for
    select ID, name
    from student
    where tot_cred > :credit_amount
    END_EXEC
    ```
The open statement for our example is as follows:

```sql
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.

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

Repeated calls to fetch get successive tuples in the query result.
A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available.

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

```
EXEC SQL close c;
```

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

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

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

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

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

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

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

```sql
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
```

- The function `dept_count` can be used to find the department names and budget of all departments with more than 12 instructors.

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

- The SQL standard supports functions that can return tables as results; such functions are called **table functions**.

- Example: Return all instructors in a given department

  ```sql
  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

  ```sql
  select *
  from table (instructor_of ('Music'))
  ```
Language Constructs (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
    where dept_name = 'Music'
do
    set n = n + r.budget
end for
```
External Language Routines

- SQL allows us to define functions in a programming language such as Java, C#, C or C++.
  - Can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.

- Declaring external language procedures and functions

```sql
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'
```
To deal with security problems, we can do one 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.
Triggers
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 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
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 (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
Recursive Queries
Recursion in SQL

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

```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_rereq, prereq
    where rec_prereq.prereq_id = prereq.course_id
)
select *
from rec_prereq;
```

This example view, `rec_prereq`, is called the *transitive closure* of the `prereq` relation.
The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of \textit{prereq} with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
  - Alternative: write a procedure to iterate as many times as required
    - See procedure \textit{findAllPrereqs} in book
Example of Fixed-Point Computation

<table>
<thead>
<tr>
<th>course_id</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>BIO-399</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>CS-190</td>
</tr>
<tr>
<td>CS-319</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-319</td>
<td>CS-315</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-319</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>Tuples in c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(CS-319)</td>
</tr>
<tr>
<td>2</td>
<td>(CS-319), (CS-315), (CS-101)</td>
</tr>
<tr>
<td>3</td>
<td>(CS-319), (CS-315), (CS-101), (CS-190)</td>
</tr>
<tr>
<td>4</td>
<td>(CS-319), (CS-315), (CS-101), (CS-190)</td>
</tr>
<tr>
<td>5</td>
<td>done</td>
</tr>
</tbody>
</table>
Advanced Aggregation Features
 Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation
  
  ```
  student_grades(ID, GPA)
  ```
  giving the grade-point average of each student
- Find the rank of each student.
- ```
  select ID, rank() over (order by GPA desc) as s_rank
  from student_grades
  ```
- An extra `order by` clause is needed to get them in sorted order
  ```
  select ID, rank() over (order by GPA desc) as s_rank
  from student_grades
  order by s_rank
  ```
- Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  - `dense_rank` does not leave gaps, so next dense rank would be 2
Ranking

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

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

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

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

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

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

```sql
select ID,
       rank() over (order by GPA desc nulls last) as s_rank
from student_grades
```
Ranking (Cont.)

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

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

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

**Window specification** in SQL:

- Given relation `sales(date, value)`

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

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

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

  ```sql
  select account_number, date_time,
         sum (value) over
         (partition by account_number
          order by date_time
          rows unbounded preceding)
  as balance
  from transaction
  order by account_number, date_time
  ```
OLAP
Data Analysis and OLAP

- **Online Analytical Processing (OLAP)**
  - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)

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

<table>
<thead>
<tr>
<th>item_name</th>
<th>color</th>
<th>clothes_size</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>skirt</td>
<td>dark</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>skirt</td>
<td>dark</td>
<td>medium</td>
<td>5</td>
</tr>
<tr>
<td>skirt</td>
<td>dark</td>
<td>large</td>
<td>1</td>
</tr>
<tr>
<td>skirt</td>
<td>pastel</td>
<td>small</td>
<td>11</td>
</tr>
<tr>
<td>skirt</td>
<td>pastel</td>
<td>medium</td>
<td>9</td>
</tr>
<tr>
<td>skirt</td>
<td>pastel</td>
<td>large</td>
<td>15</td>
</tr>
<tr>
<td>skirt</td>
<td>white</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>skirt</td>
<td>white</td>
<td>medium</td>
<td>5</td>
</tr>
<tr>
<td>skirt</td>
<td>white</td>
<td>large</td>
<td>3</td>
</tr>
<tr>
<td>dress</td>
<td>dark</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>dress</td>
<td>dark</td>
<td>medium</td>
<td>2</td>
</tr>
<tr>
<td>dress</td>
<td>dark</td>
<td>large</td>
<td>12</td>
</tr>
<tr>
<td>dress</td>
<td>pastel</td>
<td>small</td>
<td>4</td>
</tr>
<tr>
<td>dress</td>
<td>pastel</td>
<td>medium</td>
<td>3</td>
</tr>
<tr>
<td>dress</td>
<td>pastel</td>
<td>large</td>
<td>3</td>
</tr>
<tr>
<td>dress</td>
<td>white</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>dress</td>
<td>white</td>
<td>medium</td>
<td>3</td>
</tr>
<tr>
<td>dress</td>
<td>white</td>
<td>large</td>
<td>0</td>
</tr>
<tr>
<td>shirt</td>
<td>dark</td>
<td>small</td>
<td>2</td>
</tr>
<tr>
<td>shirt</td>
<td>dark</td>
<td>medium</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Cross Tabulation of sales by item_name and color

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.
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.

![Data Cube Diagram](image)
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 |

<table>
<thead>
<tr>
<th>category</th>
<th>item_name</th>
<th>dark</th>
<th>pastel</th>
<th>white</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>womenswear</td>
<td>skirt</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>dress</td>
<td>20</td>
<td>20</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>subtotal</td>
<td>28</td>
<td>28</td>
<td>15</td>
<td>88</td>
</tr>
<tr>
<td>menswear</td>
<td>pants</td>
<td>14</td>
<td>14</td>
<td>28</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>shirt</td>
<td>20</td>
<td>20</td>
<td>5</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>subtotal</td>
<td>34</td>
<td>34</td>
<td>33</td>
<td>76</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td>62</td>
<td>62</td>
<td>48</td>
<td>164</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>item_name</th>
<th>color</th>
<th>clothes_size</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>skirt</td>
<td>dark</td>
<td>all</td>
<td>8</td>
</tr>
<tr>
<td>skirt</td>
<td>pastel</td>
<td>all</td>
<td>35</td>
</tr>
<tr>
<td>skirt</td>
<td>white</td>
<td>all</td>
<td>10</td>
</tr>
<tr>
<td>skirt</td>
<td>all</td>
<td>all</td>
<td>53</td>
</tr>
<tr>
<td>dress</td>
<td>dark</td>
<td>all</td>
<td>20</td>
</tr>
<tr>
<td>dress</td>
<td>pastel</td>
<td>all</td>
<td>10</td>
</tr>
<tr>
<td>dress</td>
<td>white</td>
<td>all</td>
<td>5</td>
</tr>
<tr>
<td>dress</td>
<td>all</td>
<td>all</td>
<td>35</td>
</tr>
<tr>
<td>shirt</td>
<td>dark</td>
<td>all</td>
<td>14</td>
</tr>
<tr>
<td>shirt</td>
<td>pastel</td>
<td>all</td>
<td>7</td>
</tr>
<tr>
<td>shirt</td>
<td>White</td>
<td>all</td>
<td>28</td>
</tr>
<tr>
<td>shirt</td>
<td>all</td>
<td>all</td>
<td>49</td>
</tr>
<tr>
<td>pant</td>
<td>dark</td>
<td>all</td>
<td>20</td>
</tr>
<tr>
<td>pant</td>
<td>pastel</td>
<td>all</td>
<td>2</td>
</tr>
<tr>
<td>pant</td>
<td>white</td>
<td>all</td>
<td>5</td>
</tr>
<tr>
<td>pant</td>
<td>all</td>
<td>all</td>
<td>27</td>
</tr>
<tr>
<td>all</td>
<td>dark</td>
<td>all</td>
<td>62</td>
</tr>
<tr>
<td>all</td>
<td>pastel</td>
<td>all</td>
<td>54</td>
</tr>
<tr>
<td>all</td>
<td>white</td>
<td>all</td>
<td>48</td>
</tr>
<tr>
<td>all</td>
<td>all</td>
<td>all</td>
<td>164</td>
</tr>
</tbody>
</table>
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:

\[
\text{select } item\_name, color, size, \text{sum}(number) \\
\text{from } sales \\
\text{group by cube}(item\_name, color, size)
\]

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

  ```sql
  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.

  ```sql
  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
    ```sql
    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.,

  ```sql
  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

  ```sql
  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.,

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

  generates the groupings

  \[
  \{ \text{item\_name, ()} \} \times \{(\text{color, size}), (\text{color}), ()\}
  \]

  \[
  = \{(\text{item\_name, color, size}), (\text{item\_name, color}), (\text{item\_name}),
  (\text{color, size}), (\text{color}), (\, )\}
  \]
Online Analytical Processing Operations

- **Pivoting**: changing the dimensions used in a cross-tab is called

- **Slicing**: creating a cross-tab for fixed values only
  - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.

- **Rollup**: moving from finer-granularity data to a coarser granularity

- **Drill down**: The opposite operation - that of moving from coarser-granularity data to finer-granularity data
OLAP Implementation

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

- Early OLAP systems precomputed all possible aggregates in order to provide online response
  - Space and time requirements for doing so can be very high
    - \(2^n\) combinations of group by
  - It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
    - Can compute aggregate on \((item\_name, color)\) from an aggregate on \((item\_name, color, size)\)
      - For all but a few “non-decomposable” aggregates such as median
      - is cheaper than computing it from scratch
- Several optimizations available for computing multiple aggregates
  - Can compute aggregate on \((item\_name, color)\) from an aggregate on \((item\_name, color, size)\)
  - Can compute aggregates on \((item\_name, color, size)\), \((item\_name, color)\) and \((item\_name)\) using a single sorting of the base data
End of Chapter 5