Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #3

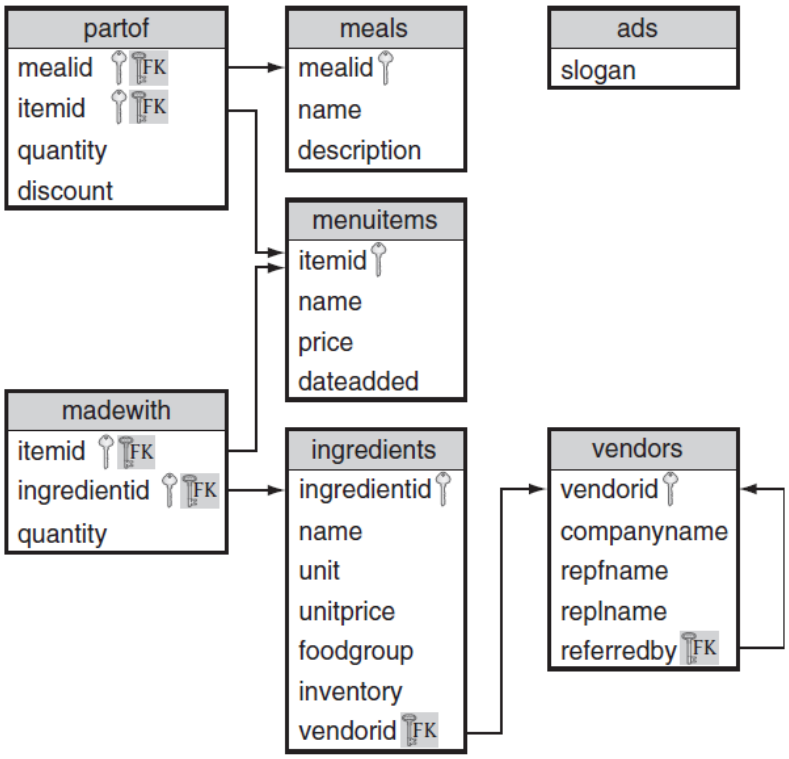
Data Manipulation Language(DML)

In the last lab, delete, update and insert queries were discussed. In today's lab, data retrieval (SELECT) queries will be discussed.

Data Retrieval

- For retrieval, SELECT, WHERE, FROM, GROUP BY, HAVING clauses are used extensively. We will practice some SQL queries on a schema of a restaurant chain. The schema has eight tables and one view**.** Run the file restaurant.sql to create the restaurant database and populate with sample data. Kindly download it from website. Save the file to local drive. In the SQL worksheet, use the command like @"D:\ restaurant.sql" to run the script. Otherwise, you can open the file using File menu also.

- The table schema is given below.



- Identify the referential integrity constraints from the schema.

SELECT, FROM, WHERE clauses:

- The SELECT clause is used to select data from a database. The SELECT clause is a query expression that begins with the SELECT keyword and includes a number of elements that form the expression. WHERE clause is used to specify the Search Conditions. The operators commonly used for comparison are =, > , < , >=, <= , <>.

Create a value menu of all items costing $0.99 or less.

```
SELECT name
       FROM items
       WHERE  PRICE <= 0.99
```

The following operators are also used in WHERE clause.

BETWEEN	Between two values(inclusive)	Vendorid BETWEEN 2 AND 10
IS NULL	Value is Null	Referredby IS NULL
LIKE	Equal Sting using wilds cards('%','_')	Name LIKE 'pri%'
IN	Equal to any element in list	Name IN ('soda',' water')
NOT	Negates a condition	NOT item IN ('GDNSD','CHKSD')

Find the food items added after 1999

```
SELECT *
     FROM items
      WHERE dateadded > '1999-12-31';
```

- * is shorthand for including all columns in the result.

Find all items with a name less than or equal to 'garden'

```
SQL>SELECT name
      FROM items
      WHERE name <= 'garden';
```

Matching String Patterns with LIKE

Wildcard	description
%	matches any substring containing 0 or more characters
_	matches any single character

Find the list of vendor representative first names that begin with 's'

```
SQL>SELECT repfname
     FROM vendors
     WHERE repfname LIKE 's%';
```

Find all vendor names containing an '_'.

```
SQL>SELECT companyname
```

```
FROM vendors
WHERE companyname LIKE '%#_%' ESCAPE '#';
```

- Note: Here _ is special wildcard character. To escape it # is used.

```
SQL>SELECT companyname
     FROM vendors
     WHERE companyname LIKE '%\_%' ESCAPE '\';
```

- Note: To escape ' we need to use one more ' before it.

Using Logical operators-AND, OR, NOT

Find the name of all of the food items other than salads.

```
SQL> SELECT name
        FROM  items
        WHERE NOT name LIKE '%Salad';
```

Find all of the ingredients from the fruit food group with an inventory greater than 100

```
SQL>SELECT ingredientid,name
      FROM ingredients
      WHERE foodgroup = 'Fruit' AND inventory >100;
```

NOTE: The precedence order is NOT, AND, OR from highest to lowest.

Find the food items that have a name beginning with either F or S that cost less than $3.50

```
SELECT name, price from items where name like 'F%' or name like 'S%'
and price < 3.50
```

- See the result of the above query. Is it correct? Modify the query to get the correct result.

BETWEEN (inclusive)

Find the food items costing between $2.50 and $3.50.

```
SQL> SELECT *
     FROM items
     WHERE price BETWEEN 2.50 AND 3.50;
```

Selecting a set of values using IN, NOT IN

Find the ingredient ID, name, and unit of items not sold in pieces or strips.

```
SQL> SELECT ingredientid,name,unit
       FROM  ingredients
       WHERE unit NOT IN  ('piece','stripe');
```

IS NULL

- SQL interprets NULL as unknown. If we compare something that is unknown to any value, even unknown, the result is unknown.
- How does SQL handle unknown? SQL only reports the rows for which the WHERE condition evaluates to true. Rows evaluating to false or unknown are not included in the answer.

Find the details of all vendors not referred by anyone.

```
SQL>SELECT *
FROM vendors
WHERE  referredby = NULL;
```

Check the output and try the following.

```
SQL>SELECT *
FROM vendors
WHERE  referredby IS NULL;
```

Do you see the difference? Comparing NULL with any attribute gives an unknown result.

Check the output of the following query

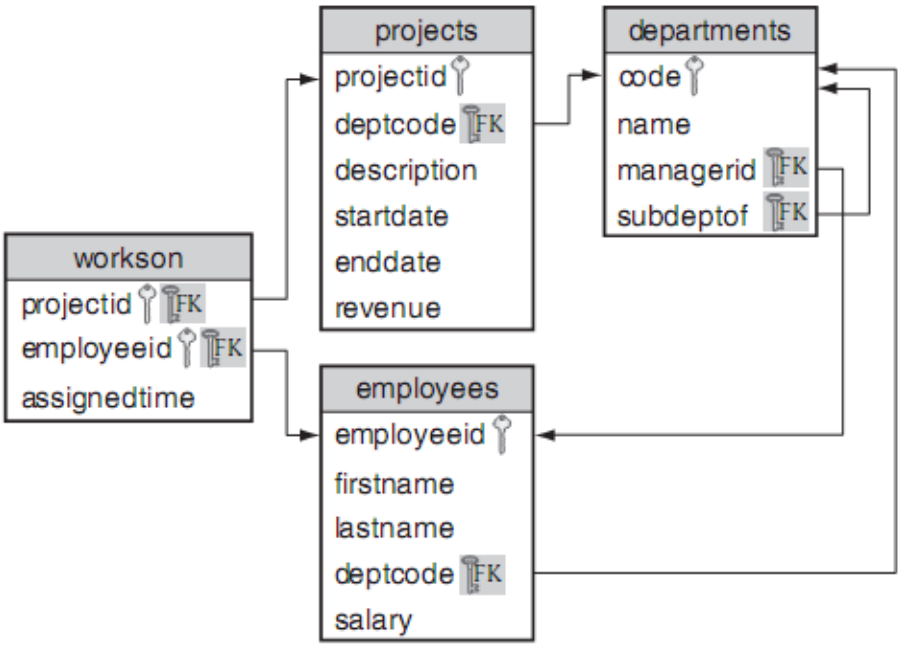
```
SQL>SELECT *
FROM items
WHERE  PRICE <= 0.99
```

EXERCISES

- Write a single SQL query for each of the following based on employees database. If you have not created the employee database in the last lab, you can create and insert data using the following .sql file. Kindly download it from website.

employeedb.sql

- The schema for the employee database is as shown below.



1. List the first and last names of all employees.
2. List all attributes of the projects with revenue greater than $40,000.
3. List the department codes of the projects with revenue between $100,000 and $150,000.
4. List the project IDs for the projects that started on or before July 1, 2004.
5. List the names of the departments that are top level (i.e., not a sub department).
6. List the ID and descriptions of the projects under the departments with code ACCNT, CNSLT, or HDWRE.
7. List all of the information about employees with last names that have exactly 8 characters and end in 'ware'.
8. List the ID and last name of all employees who work for department ACTNG and make less than $30,000.
9. List the "magical" projects that have not started (indicated by a start date in the future or *NULL)* but are generating revenue.
10. List the IDs of the projects either from the ACTNG department or that are ongoing (i.e., *NULL* end date). Exclude any projects that have revenue of $50,000 or less.

Reshaping Results

- As we know that every query result is a relation. SQL allows us to specify how this relation should be shown in a desired way. The resulting relation columns can be renamed, duplicates can be eliminated, derived attributes can be added, the rows can be sorted by some criteria etc.

- Result table columns have the same name as attributes in the original table; however you can change the result table column using a column alias using 'AS'

```
Select companyname as "Company", repfname as "First Name" from vendors;
```

- DISTINCT keyword when used in SELECT clause removes the duplicates. It can be used only once in the SELECT clause. It treats NULL as a distinct value.

Find the distinct list of food groups provided by each vendor

```
Select distinct foodgroup, vendorid from ingredients;
```

- ALL is used to specify that the result table should include duplicates also. Since by default duplicate elimination is not done, it is unnecessary.
- Data in the original table can be somewhat raw. SQL provides the ability to create derived attributes in our result table that are derived using operations and functions over existing attributes and literals.

Find the value of your pickle inventory if you double your stock of pickles.

```
SELECT ingredientid, inventory * 2 * unitprice as "Inventory Value"
from ingredients where name='Pickle'
```

- It is also possible to concatenate two or more columns in original table as one single column in result table.

Create a mailing label for each store

```
Select manager, to_char(sysdate,'dd-mm-yyyy') as 'As on', address || '
' || city || ' ' || state || ' '|| zip || 'USA' as mail from stores;
```

- Note that system date can be printed using the function *sysdate*

ORDER BY: The ORDER BY keyword is used to sort the result-set by a specified column in ASCending or DESCending order.

- The ORDER BY clause takes the output from the SELECT clause and orders the query results according to the specifications within the ORDER BY clause
- The ORDER BY keyword sort the records in ascending order by default.

Find all items from most to least expensive

```
SQL>SELECT name,price
FROM items
ORDER BY price ASC;
```

Find the name and inventory value of all ingredients ordered by inventory value

```
SQL>SELECT name,inventory*unitprice AS value
FROM ingredients
ORDER BY value DESC;
```

CASE, COALESCE, and NULLIF: Conditional Expressions

- SQL also provides basic conditional constructs to determine the correct result. CASE provides a general mechanism for specifying conditional results. SQL also provides the COALESCE and NULLIF statements to deal with NULL values, but these have equivalent CASE statements.

CASE: ValueList

- The simplest form of the CASE statement determines if a value matches any values from a list and returns the corresponding result.

```
CASE <targetexpression>
WHEN <candidateexpression> THEN <resultexpression>
```

```
WHEN <candidateexpression> THEN <resultexpression>
...
WHEN <candidateexpression> THEN <resultexpression>
[ELSE <resultexpression>]
END
```

- CASE finds the first WHEN clause where <candidateexpression> = <targetexpression> and returns the value of the corresponding <resultexpression>. If no matches are found, the value of the <resultexpression> for the ELSE clause is returned. If the ELSE clause is not specified, an implicit ELSE NULL is added to the CASE statement

Assigning goodness values for the food groups.

```
SELECT name,
CASE foodgroup
WHEN 'Vegetable' THEN 'Good'
WHEN 'Fruit' THEN 'Good'
WHEN 'Milk' THEN 'Acceptable'
WHEN 'Bread' THEN 'Acceptable'
WHEN 'Meat' THEN 'Bad'
END AS quality
FROM ingredients;
```

CASE: Conditional List

- CASE also provides another powerful format where one can specify the conditional expressions in WHEN expressions.

```
CASE
WHEN Boolean_expression1 THEN expression1
[[WHEN Boolean_expression2 THEN expression2] [...]]
[ELSE expression]
END
```

- To show the power of the CASE statement, let's put together an order for ingredients. The amount that we want to order is based on the current inventory. If that inventory is below a threshold, then we want to place an order to raise it to the threshold; otherwise, we want to order a percentage of our inventory. The exact amount is based on the type of the food item because some will spoil more quickly than others.

```
SELECT name,
FLOOR(
      CASE
            WHEN inventory < 20 THEN 20 - inventory
            WHEN foodgroup = 'Milk' THEN inventory * 0.05
            WHEN foodgroup IN ('Meat', 'Bread') THEN inventory * 0.10
```

```
            WHEN foodgroup = 'Vegetable' AND unitprice <= 0.03 THEN inventory *
            0.10
            WHEN foodgroup = 'Vegetable' THEN inventory * 0.03
            WHEN foodgroup = 'Fruit' THEN inventory * 0.04
            WHEN foodgroup IS NULL THEN inventory * 0.07
            ELSE 0
      END) AS size, vendorid
FROM ingredients
WHERE inventory < 1000 AND vendorid IS NOT NULL
ORDER BY vendorid, size;
```

NULLIF

- NULLIF takes two values and returns NULL if they are equal or the first value if the two values are not equal. You can think of it as "NULL IF equal."

```
NULLIF(<value1>,<value2>)
```

- The World Health Organization (WHO) has declared that Meat is no longer a food group.

```
SELECT ingredientid, name, unit, unitprice,
NULLIF(foodgroup, 'Meat') AS foodgroup, inventory, vendorid
FROM ingredients;
```

COALESCE

- COALESCE takes a list of values and returns the first non-NULL value.

```
COALESCE(<value1>, <value2>, : : :, <valueN>)
```

- One practical use for COALESCE is providing a substitute for NULL values in the results. For example, if we want to display all of our items with a price, we would have to handle the ones with a NULL price.

```
SELECT name, price, COALESCE(price, 0.00) AS "no nulls"
FROM items;
```

EXERCISES

Write a single SQL query for each of the following, based on employees database.

1. List all employee names as one field called name.
2. List all the department codes assigned to a project. Remove all duplicates.
3. Find the project ID and duration of each project.
4. Find the project ID and duration of each project. If the project has not finished, report its execution time as of now. [Hint: Getdate() gives current date]
5. For each completed project, find the project ID and average revenue per day.
6. Find the years a project started. Remove duplicates.
7. Find the IDs of employees assigned to a project that is more than 20 hours per week. Write three queries using 20, 40, and 60 hour work weeks.
8. For each employee assigned to a task, output the employee ID with the following:
    - 'part time' if assigned time is < 0.33
    - 'split time' if assigned time is >= 0.33 and < 0.67

- 'full time' if assigned time is >= 0.67

9. We need to create a list of abbreviated project names. Each abbreviated name concatenates the first three characters of the project description, a hyphen, and the department code. All characters must be uppercase (e.g., EMP-ADMIN).
10. For each project, list the ID and year the project started. Order the results in ascending order by year.
11. If every employee is given a 5% raise, find the last name and new salary of the employees who will make more than $50,000.
12. For all the employees in the HDWRE department, list their ID, first name, last name, and salary after a 10% raise. The salary column in the result should be named Next Year.
13. Create a neatly formatted directory of all employees, including their department code and name. The list should be sorted first by department code, then by last name, then by first name.

SQL Aggregate Functions:

- SQL aggregate functions return a single value, calculated from values in a column. The aggregate function when used in select clause, computation applies to set of all rows selected. They are commonly used in combination with GROUP BY and HAVING clauses.

Function	Returns	Data type	NULLs ignored?	DISTINCT meaningful?
AVG()	Average expression value	Numeric	Yes	Yes
MAX()	Largest expression value	Any	Yes	No
MIN()	Smallest expression value	Any	Yes	No
SUM()	Sum of expression values	Numeric	Yes	Yes
COUNT(*)	Number of rows	Any	No	Illegal
COUNT()	Number of non-null values	Any	Yes	Yes

Find the average and total price for all items

```
SQL>SELECT AVG(price), SUM(price)
FROM items;
```

Find the total number of ingredient units in inventory

```
SQL>SELECT SUM(inventory) AS totalinventoy
FROM ingredients;
```

Find the date on which last item was added

```
SQL>SELECT MAX (dateadded) AS lastmenuitem
FROM items;
```

Removing Repeating Data with DISTINCT before Aggregation

- How do aggregate functions handle repeated values? By default, an aggregate function includes all rows, even repeats, with the noted exceptions of NULL. We can add the DISTINCT qualifier to remove duplicates prior to computing the aggregate function.

Find the number of ingredients with a non-NULL food group and the number of distinct non-NULL food groups?

```
SELECT COUNT(foodgroup) AS "FGIngreds", COUNT(DISTINCT foodgroup) AS
"NoFGs" FROM ingredients;
```

Mixing Attributes, Aggregates, and Literals

- Aggregate functions return a single value, so we usually cannot mix attributes and aggregate functions in the attribute list of a SELECT statement.

```
SELECT itemid, AVG(price)
FROM items;
```

- The above query results in error. Because there are many item IDs and only one average price, SQL doesn't know how to pair them.

- The following query will mix literals and aggregates.

```
SELECT 'Results: ' AS " ", COUNT(*) AS noingredients,
COUNT(inventory) AS countedingredients,
SUM(DISTINCT inventory) AS totalingredients
FROM ingredients;
```

- Note that each aggregate function operates independently. COUNT(*) counts all rows, COUNT(quantity) counts all rows with non-NULL inventory quantities, and SUM(DISTINCT inventory) sums all rows with non-NULL and ignores repeated values.

Find the store id and total sales from each store.

```
SQL>SELECT storeid, SUM(price)
FROM orders;
```

- Any error? Can you find the reason for the error?

Group Aggregation Using GROUP BY

- Aggregate functions return a single piece of summary information about an entire set of rows. What if we wanted the aggregate over different groups of data?

GROUP BY Clause :

- The GROUP BY clause is used to group together types of information in order to summarize related data. The GROUP BY clause can be included in a SELECT statement whether or not the WHERE clause is used.
- SQL uses the GROUP BY clause to specify the attribute(s) that determine the grouping.
- Correct the above query.

Find the store id and total sales from each store.

```
SQL>SELECT storeid, SUM(price)
FROM orders
GROUP BY storeid;
```

- The GROUP BY is executed before the SELECT. Thus, the orders table is divided into groups based on the storeid values. For each of these groups, we then apply the SELECT, including the aggregate function, which in this case is SUM.

Find the total for each order.

```
SQL>SELECT storeid,ordernumber,SUM(price)
FROM orders
GROUP BY storeid,ordernumber;
```

- In this example, a group is formed for each storeid/ordernumber pair, and the aggregate is applied just as before.
- It is important to note that when we use a GROUP BY clause, we restrict the attributes that can appear in the SELECT clause. If a GROUP BY clause is present, the SELECT clause may only contain attributes appearing in the GROUP BY clause, aggregate functions (on any attribute), or literals.

Find distinct food groups supplied by each vendor.

```
SELECT vendorid, foodgroup
FROM ingredients
GROUP BY vendorid, foodgroup;
```

- GROUP BY does not require the use of aggregation functions in the attribute list. Without aggregation functions, GROUP BY acts like DISTINCT, forming the set of unique groups over the given attributes.

Removing Rows before Grouping with WHERE

- We may want to eliminate some rows from the table before we form groups. We can eliminate rows from groups using the WHERE clause.

Find the number of nonbeverages sold at each store?

```
SELECT storeid, COUNT(*)
FROM orders
WHERE menuitemid NOT IN ('SODA','WATER')
GROUP BY storeid;
```

Sorting Groups with ORDER BY

- We can order our groups using ORDER BY. It works the same as ORDER BY without grouping except that we can now also sort by group aggregates. The aggregate we use in our sort criteria need not be an aggregate from the SELECT list.

Find stores and store sales sorted by number items sold

```
SELECT storeid, SUM(price)
FROM orders
GROUP BY storeid
ORDER BY COUNT(*);
```

Removing Groups with HAVING

- Use the HAVING clause to specify a condition for groups in the final result. This is different from WHERE, which removes rows before grouping. Groups for which the HAVING condition does not evaluate to true are eliminated. Because we're working with groups of rows, it makes sense to allow aggregate functions in a HAVING predicate.

HAVING: The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

- If a GROUP BY clause is specified, the HAVING clause is applied to the groups created by the GROUP BY clause.
- If a WHERE clause is specified and no GROUP BY clause is specified, the HAVING clause is applied to the output of the WHERE clause and that output is treated as one group.
- If no WHERE clause and no GROUP BY clause are specified, the HAVING clause is applied to the output of the FROM clause and that output is treated as one group.

Find the maximum number of items in an order for each store with total sales of more than $20.

```
SQL>SELECT storeid, MAX(linenumber) AS "Items Sold"
FROM orders
GROUP BY storeid
HAVING SUM(price)>20;
```

Find total sales at FIRST store

```
SELECT SUM(price) AS sales
FROM orders
GROUP BY storeid
HAVING storeid='FIRST';
```

- Above query could also written using WHERE clause. That would be faster.
- The HAVING clause can use AND, OR, and NOT just like the WHERE clause to form more complicated predicates on groups.

Find the minimum and maximum unit price of all ingredients in each non-NULL food group. The results are only reported for food groups with either two or more items or a total inventory of more than 500 items.

```
SELECT foodgroup, MIN(unitprice) AS minprice, MAX(unitprice) AS
maxprice
FROM ingredients
WHERE foodgroup IS NOT NULL
GROUP BY foodgroup
HAVING COUNT(*) >= 2 OR SUM(inventory) > 500;
```

EXERCISES on restaurant database

1. Find the ID of the vendor who supplies grape
2. Find all of the ingredients from the fruit food group with an inventory greater than 100
3. Display all the food groups from ingredients, in which 'grape' is not a member.
4. Find the ingredients, unit price supplied by 'VGRUS'(vendor ID) order by unit price(asc)
5. Find the date on which the last item was added.
6. Find the number of vendors each vendor referred, and only report the vendors reffering more than one.

EXERCISES

Write a single SQL query for each of the following, based on employees database.

1. Find the average salary for all employees.
2. Find the minimum and maximum project revenue for all active projects that make money.
3. Find the number of projects that are completed. You may not use a WHERE clause.
4. Find the number of projects that have been worked on or currently are being worked on by an employee.
5. Find the last name of the employee whose last name is last in dictionary order.
6. Compute the employee salary standard deviation. As a reminder, the formula for the population standard deviation is as follows:

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

7. Find the number of employees who are assigned to some department. You may not use a WHERE clause.
8. For each department, list the department code and the number of employees in the department.
9. For each department that has a project, list the department code and report the average revenue and count of all of its projects.
10. Find the employee ID of all employees where their assigned time to work on projects is 100% or more.
11. Calculate the salary cost for each department with employees that don't have a last name ending in "re" after giving everyone a 10% raise**.**

---------&---------