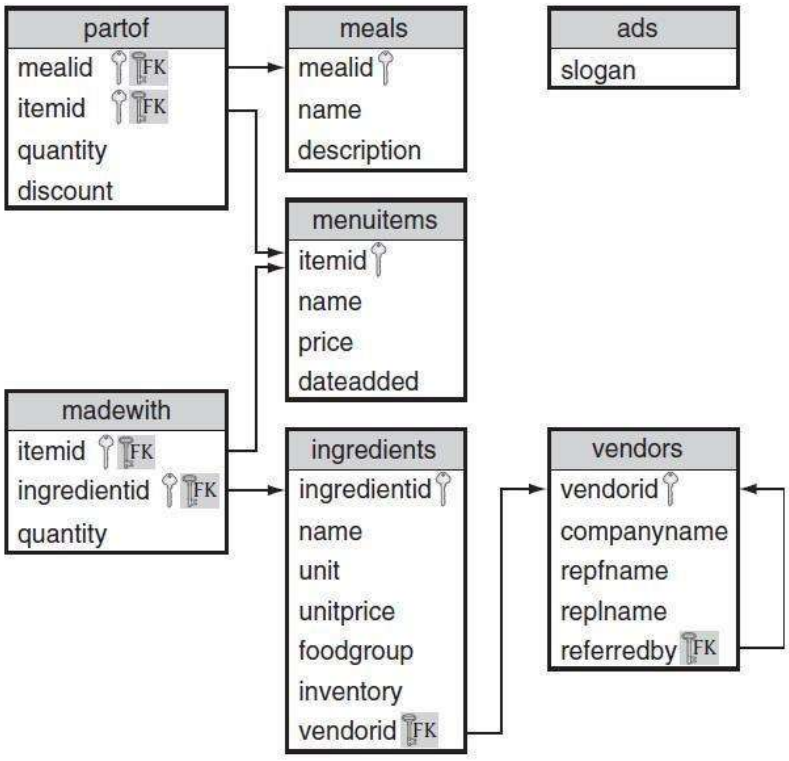
Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #3

In this lab we will continue practicing SQL queries related to aggregate functions and joins etc on a schema of a restaurant chain created and populated in the last lab. The table schema is given below.



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.

AVG()	Returns the average value
COUNT()	Returns the number of rows
MIN()	Returns the smallest value
SUM()	Returns the sum
MAX()	Returns the largest value

Find the average and total price for all items

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

*F*ind *the total number of ingredient units in inventory*

```
SELECT SUM(inventory) AS totalInventory
FROM ingredients;
```

Find the date on which last item was added

```
SELECT MAX (dateadded) AS lastMenuItem
FROM items;
```

Find the store id and total sales from all stores.

```
SELECT storeid,sum(price)
FROM orders;
```

❖ Is there any error in the last query? If so, can you find the reason for the error?

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(inventory) counts all rows with non-NULL inventory quantities, and SUM(DISTINCT inventory) sums all rows with non-NULL and ignores repeated values.

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 total for each order.

```
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.

Find the number of distinct food groups provided by each vendor.

```
SELECT vendorid,COUNT( distinct(foodgroup) )
FROM ingredients
GROUP BY vendorid;
```

Find stores and store sales sorted by number of items sold.

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

- 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.

```
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;
```

EXERCISE 1

Write a single SQL query for each of the following, based on the 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. Find the ingredients, unit price supplied by 'VGRUS'(vendor ID) order by unit price(asc)
4. Find the date on which the last item was added.
5. Find the number of vendors each vendor referred, and only report the vendors reffering more than one.

EXERCISE 2

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 average salary of employees in every department.
3. Find the minimum and maximum project revenue for all active projects that make money.
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.

JOIN

- Until now, all of our queries have used a single table. It is not surprising that many queries require information from more than one table. Suppose that you want a list of ingredients and their vendor (ID and company name).
- Combining tables to derive a new table is called a *join.*

For each ingredient, find its name and the name and ID of the vendor that supplies it

```
SELECT vendors.vendorid, name, companyname
FROM ingredients, vendors
WHERE ingredients.vendorid = vendors.vendorid;
```

Find the names of the ingredients supplied to us by Veggies_R_Us

```
SELECT name
FROM ingredients, vendors
WHERE ingredients.vendorid = vendors.vendorid AND companyname =
'Veggies_R_Us';
```

- Designing a join is a two-step process:
    1. Find the tables with the information that we need to answer the query.
    2. Determine how to connect the tables.

SELF JOIN- Joining a Table with Itself:

- SQL handles this situation by allowing two copies of the same table to appear in the FROM clause Allowing two copies of the same table to appear in the FROM clause. The only requirement is that each copy of the table must be given a distinct alias to distinguish between the copies of the table

Find all of the vendors referred by Veggies_R_Us

```
SELECT v2.companyname
FROM vendors v1, vendors v2   /*Note the table alias*/
WHERE v1.vendorid = v2.referredby AND v1.companyname = 'Veggies_R_Us';
```

THETA JOIN- Generalizing Join Predicates:

- A join where the join predicate uses any of the comparison operators is called a ***theta join***

Find all of the items and ingredients where we do not have enough of the ingredient to make three items.

```
SELECT items.name, ing.name
FROM items, madewith mw, ingredients ing
WHERE items.itemid = mw.itemid AND mw.ingredientid = ing.ingredientid
AND
3 * mw.quantity > ing.inventory;
```

Find the name of all items that cost more than the garden salad

```
SELECT a.name
FROM items a, items q
WHERE a.price > q.price AND q.name = 'Garden Salad';
```

JOIN Operators:

- Joining tables together is so common that SQL provides a JOIN operator for use in the FROM clause. There are several variants of the JOIN, which we explore here.
- **INNER JOIN**: This is the default type of the join if only the keyword 'JOIN' is used.
  INNER JOIN takes two tables and a join specification describing how the two tables should be joined. The join specification may be specified as a condition. The condition follows the keyword ON

Find the names of the ingredients supplied to us by Veggies_R_Us

```
SELECT name
FROM ingredients i INNER JOIN vendors v ON i.vendorid = v.vendorid
WHERE v.companyname = 'Veggies_R_Us';
```

- It is common to join tables over attributes with the same name.

Find the name of all items that cost more than the garden salad

```
SELECT i1.name
FROM items i1 INNER JOIN items i2 ON i1.price > i2.price
WHERE i2.name = 'Garden Salad';
```

- Can the above two queries can also be run without using JOIN keyword? How? Which way is efficient?

Find the names of items that are made from ingredients supplied by the company Veggies_R_Us

```
SELECT DISTINCT(it.name)
FROM vendors v JOIN ingredients i ON i.vendorid = v.vendorid JOIN
madewith m on
m.ingredientid = i.ingredientid JOIN
items it on m.itemid = it.itemid
WHERE v.companyname = 'Veggies_R_Us';
```

❖ **OUTER JOIN**:

Outer joins are useful where we want not only the rows that satisfy the join predicate, but also the rows that do not satisfy it. There are three types of OUTER JOIN: FULL, LEFT, and RIGHT. FULL OUTER JOIN includes three kinds of rows:

- All rows that satisfy the join predicate (same as INNER JOIN)
- All rows from the first table that don't satisfy the join predicate for any row in the second table
- All rows from the second table that don't satisfy the join predicate for any row in the first table.

For each ingredient, find its name and the name and ID of the vendor that supplies them. Include vendors who supply no ingredients and ingredients supplied by no vendors

```
SELECT companyname, i.vendorid, i.name
FROM vendors v FULL JOIN ingredients i ON v.vendorid = i.vendorid;
```

Find the vendors who do not provide us with any ingredients

```
SELECT companyname
FROM vendors v LEFT JOIN ingredients i on v.vendorid=i.vendorid
WHERE ingredientid IS NULL;
```

❖ **CROSS JOIN**:

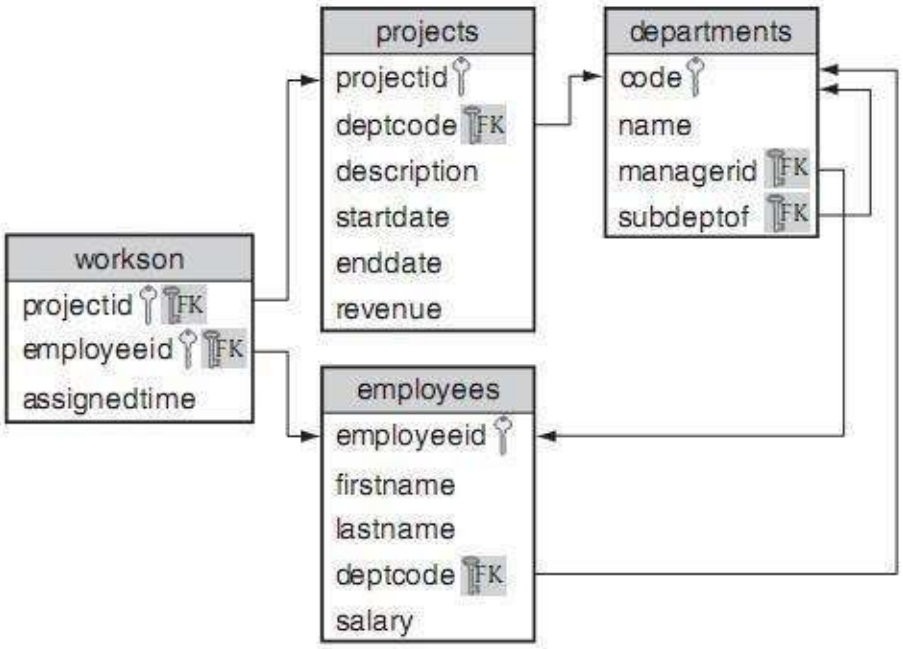
SQL also provides a CROSS JOIN. It computes the cross product of two tables. We cannot use OUTER, NATURAL, USING, or ON with CROSS JOINS. Note that this is the same behaviour as using a comma-delimited list of tables in the FROM clause. The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table.

Find all the combinations of item name and company name across two tables – vendors & ingredients

```
SELECT i.name, v.companyname AS Vendor
FROM ingredients i
CROSS JOIN vendors v
ORDER BY v.vendorid ;
```

EXERCISE 3

Write a single SQL query for each of the following based on employee database created in the last lab. The scheme for employee database is shown below.



1. Find the names of all people who work in the Consulting department. Solve it two ways: 1) using only WHERE-based join (i.e., no INNER/OUTER/CROSS JOIN) 2) with CROSS JOIN.
2. Find the names of all people who work in the Consulting department and who spend more than 20% of their time on the project with ID ADT4MFIA. Solve three ways: 1) using only WHEREbased join (i.e., no INNER/OUTER/CROSS JOIN), 2) using some form of JOIN.
3. Find the total percentage of time assigned to employee Abe Advice. Solve it two ways: 1) using only WHERE-based join (i.e., no INNER/OUTER/CROSS JOIN) and 2) using some form of JOIN.
4. Find the descriptions of all projects that require more than 70% of an employee's time. Solve it two ways: 1) using only WHERE-based join (i.e., no INNER/OUTER/CROSS JOIN) and 2) using some form of JOIN.
5. For each employee, list the employee ID, number of projects, and the total percentage of time for the current projects to which she is assigned. Include employees not assigned to any project.
6. Find the description of all projects with no employees assigned to them.
7. For each project, find the greatest percentage of time assigned to one employee. Solve it two ways: using only WHERE-based join (i.e., no INNER/OUTER/CROSS JOIN) and 2) using some form of JOIN.
8. For each employee ID, find the last name of all employees making more money than that employee. Solve it two ways: 1) using only WHERE-based join (i.e., no INNER/OUTER/CROSS JOIN) and 2) using some form of JOIN.

UNION, INTERSECTION and EXCEPT

❖ **Union**: UNION combines two query results. The syntax for UNION is as follows:

```
<left SELECT> UNION [{ALL | DISTINCT}] <right SELECT>.
```

- The <left SELECT> and <right SELECT> can be almost any SQL query, provided that the result sets from the left and right SELECT are compatible. Two result sets are compatible if they have the same number of attributes and each corresponding attribute is compatible. Two attributes are compatible if SQL can implicitly cast them to the same type.
- Out of ALL or DISTINCT options DISTINCT is default. that means query will eliminate duplicates by default.
- if we want the sets to be treated as multi sets the we should use ALL option instead.

Find the list of item prices and ingredient unit prices

```
SELECT price
            FROM items
            UNION
            SELECT unitprice
            FROM ingredients;
```

Find the names and prices of meals and items

```
            SELECT name, price
            FROM items
            UNION
            SELECT m.name, SUM(quantity * price * (1.0 - discount))
            FROM meals m, partof p, items i
            WHERE m.mealid = p.mealid AND p.itemid = i.itemid
            GROUP BY m.mealid, m.name;
```

❖ **Intersection**: The intersection of two sets is all the elements that are common to both sets. In SQL, the INTERSECT operator returns all rows that are the same in both results. The syntax for INTERSECT is as follows:

```
<left SELECT> INTERSECT [{ALL |DISTINCT}] <right SELECT>
```

o The <left SELECT> and <right SELECT> must be compatible, as with UNION . By default, INTERSECT eliminates duplicates.

o If the left and right tables have *l* and *r* duplicates of a value, v; the number of duplicate values resulting from an INTERSECT ALL is the minimum of *l* and *r*.

Find all item IDs with ingredients in both the Fruit and Vegetable food groups

```
            SELECT itemid
            FROM madewith mw, ingredients ing
            WHERE mw.ingredientid = ing.ingredientid and foodgroup = 'Vegetable'
            INTERSECT
            SELECT itemid
            FROM madewith mw, ingredients ing
            WHERE mw.ingredientid = ing.ingredientid and foodgroup = 'Fruit';
```

❖ **Difference**: Another common set operation is set difference. If R and S are sets, then R - S contains all of the elements in R that are not in S. The EXCEPT (MINUS in Oracle 11g) operator

in SQL is similar, in that it returns the rows in the first result that are not in the second one. The syntax for EXCEPT is as follows:

<left SELECT> EXCEPT [ALL | DISTINCT] <right SELECT>

- The <left SELECT> and <right SELECT> must be compatible, as with UNION and INTERSECT. Like UNION and INTERSECT, EXCEPT eliminates duplicates.

Find all item IDs of items not made with Cheese

```
SELECT itemid
FROM items
EXCEPT
SELECT itemid
FROM madewith mw, ingredients ing
WHERE mw.ingredientid = ing.ingredientid AND ing.name = 'Cheese';
```

Find all item IDs of items not made with Cheese

```
SELECT DISTINCT(itemid)
FROM madewith mw, ingredients ing
WHERE mw.ingredientid = ing.ingredientid AND ing.name != 'Cheese';
```

❖ Did you get the result? What is wrong with above query?

List all the food groups provided by some ingredient that is in the Fruit Plate but not the Fruit Salad?

```
SELECT foodgroup
FROM madewith m, ingredients i
WHERE m.ingredientid = i.ingredientid AND m.itemid = 'FRPLT'
EXCEPT
SELECT foodgroup
FROM madewith m, ingredients i
WHERE m.ingredientid = i.ingredientid AND m.itemid = 'FRTSD';
```

Find the vendors who do not provide us with any ingredients

```
SELECT companyname
FROM vendors v LEFT JOIN ingredients i on v.vendorid=i.vendorid
WHERE ingredientid IS NULL;
```

❖ Rewrite the above query to use EXCEPT instead of an OUTER JOIN

❖ From set theory, $R \cap S = R - (R - S)$. Rewrite the following query using EXCEPT instead of INTERSECT.

```
SELECT itemid
FROM madewith m, ingredients i
WHERE m.ingredientid = i.ingredientid AND foodgroup='Milk'
INTERSECT
SELECT itemid
FROM madewith m, ingredients i
WHERE m.ingredientid = i.ingredientid AND foodgroup='Fruit';
```

EXERCISE 4

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

1. Find all dates on which projects either started or ended. Eliminate any duplicate or *NULL* dates. Sort your results in descending order.
2. Display all the food groups from ingredients, in which ‘grape’ is not a member.
3. Use INTERSECT to find the first and last name of all employees who both work on the Robotic Spouse and for the Hardware department.
4. Use EXCEPT to find the first and last name of all employees who work on the Robotic Spouse but not for the Hardware department.
5. Find the first and last name of all employees who work on the Download Client project but not the Robotic Spouse project.
6. Find the first and last name of all employees who work on the Download Client project and the Robotic Spouse project.
7. Find the first and last name of all employees who work on either the Download Client project or the Robotic Spouse project.
8. Find the first and last name of all employees who work on either the Download Client project or the Robotic Spouse project but not both.
9. Using EXCEPT, find all of the departments without any projects.

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