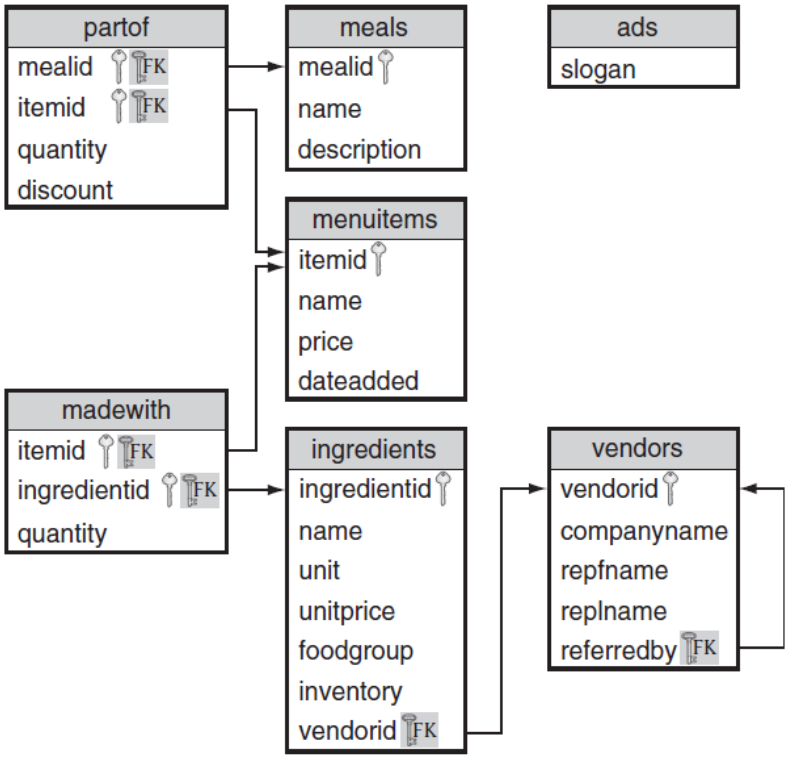
Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #5

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



Sub-queries:

- So far, we've seen joins and set operators for combining tables together. SQL provides another way to combine tables. You can nest queries within queries. Such an embedded query is called a subquery. A subquery computes results that are then used by an outer query. Basically, a subquery acts like any other expression we've seen so far. A subquery can be nested inside the SELECT, FROM, WHERE, and HAVING clauses. You can even nest subqueries inside another subquery

Find the names of the ingredients supplied by Veggies_R_Us

```
SELECT name
FROM ingredients
WHERE vendorid =
(SELECT vendorid
FROM vendors
```

```
WHERE companyname = 'Veggies_R_Us');
```

- SQL marks the boundaries of a subquery with parentheses.

- Some points to remember when using subqueries:
  - Only the columns of the outermost query can appear in the result table. When creating a new query, the outermost query must contain all of the attributes needed in the answer.
  - There must be some way of connecting the outer query to the inner query. All SQL comparison operators work with subqueries.
  - Subqueries are restricted in what they can return. First, the row and column count must match the comparison operator. Second, the data types must be compatible.

Subqueries using IN:

- In the above query, the = operator makes the connection between the queries. Because = expects a single value, the inner query may only return a result with a single attribute and row. If subquery returns multiple rows, we must use a different operator. We use the IN operator, which expects a subquery result with zero or more rows.

Find the name of all ingredients supplied by Veggies_R_Us or Spring Water Supply

```
SELECT name
FROM ingredients
WHERE vendorid IN
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us' OR
companyname = 'Spring Water Supply');
```

Find the average unit price for all items provided by Veggies_R_Us

```
SELECT AVG(unitprice) AS avgprice
FROM ingredients
WHERE vendorid IN
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us');
```

Subqueries using BETWEEN:

Find all of the ingredients with an inventory within 25% of the average inventory of ingredients.

```
SELECT name
FROM ingredients
WHERE inventory BETWEEN
(SELECT AVG(inventory) * 0.75
```

```
FROM ingredients)
AND
(SELECT AVG(inventory) * 1.25
FROM ingredients);
```

- Subqueries can even be combined with other predicates in the WHERE clause, including other Subqueries. Look at the following query.

Find the companies who were referred by Veggies_R_Us and provide an ingredient in the milk food group

```
SELECT companyname
FROM vendors
WHERE (referredby IN
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us')) AND
(vendorid IN
(SELECT vendorid
FROM ingredients
WHERE foodgroup = 'Milk'));
```

Multilevel Subquery Nesting:

Find the name and price for all items using an ingredient supplied by Veggies_R_Us

```
SELECT name, price
FROM items
WHERE itemid IN
     (SELECT itemid -- Subquery 3
     FROM madewith
     WHERE ingredientid IN
          (SELECT ingredientid -- Subquery 2
          FROM ingredients
          WHERE vendorid =
               (SELECT vendorid -- Subquery 1
               FROM vendors
               WHERE companyname = 'Veggies_R_Us')));
```

- First the innermost subquery is executed which return one single value. Then subquery2 is executed, the result of which is passed to subquery3. We can use GROUP BY, HAVING, and ORDER BY with subqueries.

For each store, find the total sales of items made with ingredients supplied by Veggies_R_Us. Ignore meals and only consider stores with at least two such items sold

```
SELECT storeid, SUM(price) AS sales
FROM orders
WHERE menuitemid IN
     (SELECT itemid -- Subquery 3
     FROM madewith
     WHERE ingredientid IN
          (SELECT ingredientid -- Subquery 2
          FROM ingredients
          WHERE vendorid =
               (SELECT vendorid -- Subquery 1
               FROM vendors
               WHERE companyname = 'Veggies_R_Us')))
GROUP BY storeid
HAVING COUNT(*) > 2
ORDER BY sales DESC;
```

- GROUP BY, HAVING can be used in subqueries as well. Using ORDER BY in a subquery makes little sense because the order of the results is determined by the execution of the outer query.

Subqueries Using NOT IN:

Find all of the ingredients supplied by someone other than Veggies_R_Us

```
SELECT name
FROM ingredients
WHERE vendorid NOT IN
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us');
```

Find the company name of the small vendors who don't provide any ingredients with large (>100) inventories

```
SELECT companyname
FROM vendors
WHERE vendorid NOT IN
(SELECT vendorid
FROM ingredients
WHERE inventory > 100);
```

- By examining the ingredients table, we can see that Flavorful Creams ought to be the answer. What happened? The ingredient Secret Dressing does not have a vendor, so the list of values contains a NULL. How do we fix this?

- **Solution**: Eliminate *NULL* values from the subquery result. Rewrite the above query!!!

```
SELECT companyname
FROM vendors
WHERE vendorid NOT IN
(SELECT vendorid
FROM ingredients
WHERE inventory > 100 AND vendorid IS NOT NULL);
```

- **Note**: IN over an empty table always returns false, therefore NOT IN always returns true.

Subqueries with Empty Results:

- What happens when a subquery result is empty? The answer depends on what SQL is expecting from the subquery. Let's look at a couple of examples. Remember that the standard comparison operators expect a scalar value; therefore, SQL expects that any subqueries used with these operators always return a single value. If the subquery result is empty, SQL returns NULL.

```
SELECT companyname
FROM vendors
WHERE referredby =
(SELECT vendorid
FROM vendors
WHERE companyname = 'No Such Company');
```

- Here the subquery results are empty so the subquery returns NULL. Evaluating the outer query, referredby = NULL returns unknown for each row in vendors. Consequently, the outer query returns an empty result table.

- When using the IN operator, SQL expects the subquery to return a table. When SQL expects a table from a subquery and the subquery result is empty, the subquery returns an empty table. IN over an empty table always returns false, therefore NOT IN always returns true.

```
SELECT companyname
FROM vendors
WHERE referredby NOT IN
(SELECT vendorid
FROM vendors
WHERE companyname = 'No Such Company');
```

- The above subquery returns a table with zero rows (and one column). For every row in the vendors table, NOT IN returns true over the subquery, so every row is returned.

Combining JOIN and Subqueries:

- Nested queries are not restricted to a single table.

Find the name and price of all items using an ingredient supplied by Veggies_R_Us

```
SELECT DISTINCT items.itemid, price
FROM items JOIN madewith ON items.itemid=madewith.itemid
WHERE ingredientid IN
(SELECT ingredientid
FROM ingredients JOIN vendors on vendors.vendorid=ingredients.vendorid
WHERE companyname = 'Veggies_R_Us');
```

Standard Comparison Operators with Lists Using ANY, SOME, or ALL

- We can modify the meaning of the SQL standard comparison operators with ANY, SOME, and ALL so that the operator applies to a list of values instead of a single value.
- Using ANY or SOME:
  - The ANY or SOME modifiers determine if the expression evaluates to true for at least one row in the subquery result.

Find all items that have a price that is greater than any salad item

```
SELECT name
FROM items
WHERE price > ANY
(SELECT price
FROM items
WHERE name LIKE '%Salad');
```

Find all ingredients not supplied by Veggies_R_Us or Spring Water Supply

```
SELECT name
FROM ingredients
WHERE ingredientid NOT IN
(SELECT ingredientid
FROM ingredients
WHERE vendorid = ANY
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us' OR
companyname = 'Spring Water Supply'));
```

  - Be aware of the difference between <> ANY and NOT IN. `x <> ANY y` returns true if any of the values in y are not equal to x. x NOT IN y returns true only if none of the values in y are equal to x or if the list y is empty

- Using ALL: The ALL modifier determines if the expression evaluates to true for *all* rows in the subquery result.

Find all ingredients that cost at least as much as every ingredient in a salad

```
SELECT name
FROM ingredients
WHERE unitprice >= ALL
(SELECT unitprice
FROM ingredients ing JOIN madewith mw on
mw.ingredientid=ing.ingredientid JOIN
items i on mw.itemid=i.itemid WHERE i.name LIKE '%Salad');
```

Find the name of all ingredients supplied by someone other than Veggies_R_Us or Spring Water Supply

```
SELECT name
FROM ingredients
WHERE vendorid <> ANY
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us' OR
companyname = 'Spring Water Supply');
```

- Correct query:

```
SELECT name
FROM ingredients
WHERE ingredientid NOT IN
(SELECT ingredientid
FROM ingredients
WHERE vendorid = ANY
(SELECT vendorid
FROM vendors
WHERE companyname = 'Veggies_R_Us' OR
companyname = 'Spring Water Supply'));
```

Find the most expensive items

```
SELECT *
FROM items
WHERE price >= ALL
(SELECT price
FROM items);
```

- What is wrong with above query ? Correct query:

```
SELECT *
FROM items
WHERE price >= ALL
(SELECT max(price)
FROM items);
```

- You might be tempted to believe that >= ALL is equivalent to >= (SELECT MAX()), but this is not correct. What's going on here? Recall that for ALL to return true, the condition must be true for all rows in the subquery. NULL prices evaluate to unknown; therefore, >= ALL evaluates to unknown so the result is empty. Of course, we can solve this problem by eliminating NULL prices. However, NULL isn't the only problem. What happens when the subquery result is empty?

- A common mistake is to assume that the = ALL operator returns true if all of the rows in the outer query match all of the rows in the inner query. This is not the case. A row in the outer query will satisfy the = ALL operator only if it is equal to all of the values in the subquery. If the inner query returns multiple rows, each outer query row will only satisfy the = ALL predicate if all rows in the inner query have the same value and that value equals the outer query row value. Notice that the exact same result is achieved by using = alone and ensuring that only a distinct value is returned by the inner query.

Correlated Subqueries:

- Correlated subqueries are not independent of the outer query. Correlated subqueries work by first executing the outer query and then executing the inner query for each row from the outer query.

Find the items that contain 3 or more ingredients

```
SELECT itemid, name
FROM items
WHERE (SELECT COUNT(*)
FROM madewith
WHERE madewith.itemid = items.itemid) >= 3;
```

- Look closely at the inner query. It cannot be executed independently from the outer query because the WHERE clause references the items table from the outer query. Note that in the inner query we must use the table name from the outer query to qualify itemid. How does this execute? Because it's a correlated subquery, the outer query fetches all the rows from the items table. For each row from the outer query, the inner query is executed to determine the number of ingredients for the particular itemid.

Find all of the vendors who referred two or more vendors

```
SELECT vendorid, companyname
FROM vendors v1
```

```
WHERE (SELECT COUNT(*)
FROM vendors v2
WHERE v2.referredby = v1.vendorid) >= 2;
```

- EXISTS: EXISTS is a conditional that determines if any rows exist in the result of a subquery. EXISTS returns true if <subquery> returns at least one row and false otherwise.

Find the meals containing an ingredient from the Milk food group

```
SELECT *
FROM meals m
WHERE EXISTS
(SELECT *
FROM partof p JOIN items on p.itemid=items.itemid
JOIN madewith on items.itemid=madewith.itemid
JOIN ingredients on madewith.ingredientid=ingredients.ingredientid
WHERE foodgroup = 'Milk' AND m.mealid = p.mealid);
```

Find all of the vendors that did not recommend any other vendor

```
SELECT vendorid, companyname
FROM vendors v1
WHERE NOT EXISTS (SELECT *
FROM vendors v2
WHERE v2.referredby = v1.vendorid);
```

Derived Relations — Subqueries in the FROM Clause:

List the name and inventory value of each ingredient in the Fruit or Vegetable food group and its supplier

```
SELECT name, companyname, val
FROM vendors v,
(SELECT name, vendorid, unitprice * inventory as val FROM ingredients i
WHERE foodgroup IN ('Fruit', 'Vegetable') )  d
WHERE v.vendorid = d.vendorid
```

- The subquery generates a derived table containing the name, vendor ID, and inventory value of each ingredient in the specified food groups. Using an expanded form of table aliases, we name the derived table and its attributes. Next, SQL performs the Cartesian product of the vendors table and the derived table and applies the join predicate. Recall that the FROM clause is executed first by the DBMS. When the subquery is executed, no rows from the other tables have been retrieved. Thus, the DBMS cannot reference another table inside a subquery in the FROM clause. All derived relations are uncorrelated subqueries.

- There are two advantages of derived relations. First, it allows us to break down complex queries into easier to understand parts. The second advantage of derived relations is that we can improve the performance of some queries. If a derived relation is much smaller than the original relation, then the query may execute much faster.

Subqueries in the HAVING Clause:

- We can embed both simple and correlated subqueries in the HAVING clause. This works much like the WHERE clause subqueries, except we are defining predicates on groups rather than rows.

Find all vendors who provide more ingredients than Spring Water Supply

```
SELECT companyname
FROM vendors v, ingredients i
WHERE i.vendorid = v.vendorid
GROUP BY v.vendorid, companyname
HAVING COUNT(*) > (SELECT COUNT(*)
FROM ingredients i, vendors v
WHERE i.vendorid = v.vendorid AND
companyname = 'Spring Water Supply');
```

- Correlated subqueries in the HAVING clause allow us to evaluate per-group conditionals.

Find the average inventory values for each vendor who recommends at least one other vendor

```
SELECT v1.vendorid, AVG(unitprice * inventory)
FROM ingredients JOIN vendors v1 ON (ingredients.vendorid=v1.vendorid)
GROUP BY v1.vendorid
HAVING EXISTS (SELECT *
FROM vendors v2
WHERE v1.vendorid = v2.referredby);
```

- This query begins by grouping all of the ingredients together by vendor. Next, for each vendor, the subquery finds the vendors referred by that vendor. If the subquery returns any row, EXISTS returns true and the vendor ID and average inventory value are included in the final result.

Division Operation:

- One of the most challenging types of queries for SQL is one that compares two groups of rows to see if they are the same. These types of queries arise in many different applications. Some examples are as follows:

- Has a student taken all the courses required for graduation?
- List the departments and projects for which that department has all of the tools required for the project.
- Find all the items that contain all of the ingredients provided by a vendor.

❖ There are two approaches.
- One is using set operations
- and other using set cardinality. cardinality of a relation is the number of tuples in that set.

❖ **Using set operations:** To find whether a student has taken all courses required for graduation, let set A be courses taken by student and let set B be all courses required for graduation. If B-A is empty then, the student has taken all courses. If we need to do this for all students in a single query, we need to use correlated sub query. The query will be like

```
select idno, name
from student
where NOT exists(
(select courseno from enrollment where enrollment.idno=student.idno)
MINUS
(select courseno from graduate_requirements)
)
```

❖ Here for every student we are checking whether the difference is empty or not. Suppose the student is in different programs like A7, B5 ... then the set B will also vary.

```
select idno, name,programno
from student, programs
where NOT exists(
(select courseno from enrollment where enrollment.idno=student.idno)
MINUS
(select courseno from graduate_requirements g where
g.programno=programs.programno)
)
```

Find all items and vendors such that all ingredients supplied by that vendor are in the item.

```
SELECT name, companyname
FROM items it, vendors v
WHERE NOT EXISTS (
SELECT ingredientid -- ingredients supplied by vendor
FROM ingredients i
WHERE i.vendorid = v.vendorid
MINUS
SELECT ingredientid -- ingredients used in item
FROM madewith m
WHERE it.itemid = m.itemid);
```

Find all vendors and items such that all ingredients in the item are from the same vendor.

```
SELECT i.name, companyname
```

```
FROM items i, vendors v
WHERE NOT EXISTS (
(SELECT m.ingredientid -- ingredients used in item
FROM madewith m
WHERE i.itemid = m.itemid)
MINUS
(SELECT ingredientid -- ingredients supplied by vendors
FROM ingredients i
WHERE i.vendorid = v.vendorid));
```

- **Comparing relational cardinality**: In this case we compare whether for every combination of A and B whether the number of elements in A and B are equal or not.

Find all items and vendors such that all ingredients in the item are supplied by that vendor.

```
SELECT i.name, companyname
FROM items i, vendors v
WHERE
(SELECT COUNT(DISTINCT m.ingredientid) -- number of ingredients in item
FROM madewith m
WHERE i.itemid = m.itemid)
= -- number of ingredients in item supplied by vendor
(SELECT COUNT(DISTINCT m.ingredientid)
FROM madewith m, ingredients n
WHERE i.itemid = m.itemid AND m.ingredientid = n.ingredientid
AND n.vendorid = v.vendorid);
```

Find all items and vendors such that all ingredients supplied by that vendor are in the item.

```
SELECT name, companyname
FROM items i, vendors v
WHERE -- number of ingredients in item supplied by vendor
(SELECT COUNT(DISTINCT m.ingredientid)
FROM madewith m, ingredients ing
WHERE i.itemid = m.itemid AND m.ingredientid = ing.ingredientid AND
ing.vendorid = v.vendorid)
=
(SELECT COUNT(DISTINCT ing.ingredientid) -- number of ingredients
supplied by vendor
FROM ingredients ing
WHERE ing.vendorid = v.vendorid);
```

Subqueries in the SELECT Clause:

- We can include subqueries in the SELECT clause to compute a column in the result table. It works much like any other expression. You may use both simple and correlated subqueries in the SELECT clause.

For each ingredient, list its inventory value, the maximum inventory value of all ingredients, and the ratio of the ingredient's inventory value to the average ingredient inventory value.

```
SELECT name, unitprice * inventory AS "Inventory Value",
(SELECT MAX(unitprice * inventory)
FROM ingredients) AS "Max. Value",
(unitprice * inventory) / (SELECT AVG(unitprice * inventory)
FROM ingredients) AS "Ratio"
FROM ingredients;
```

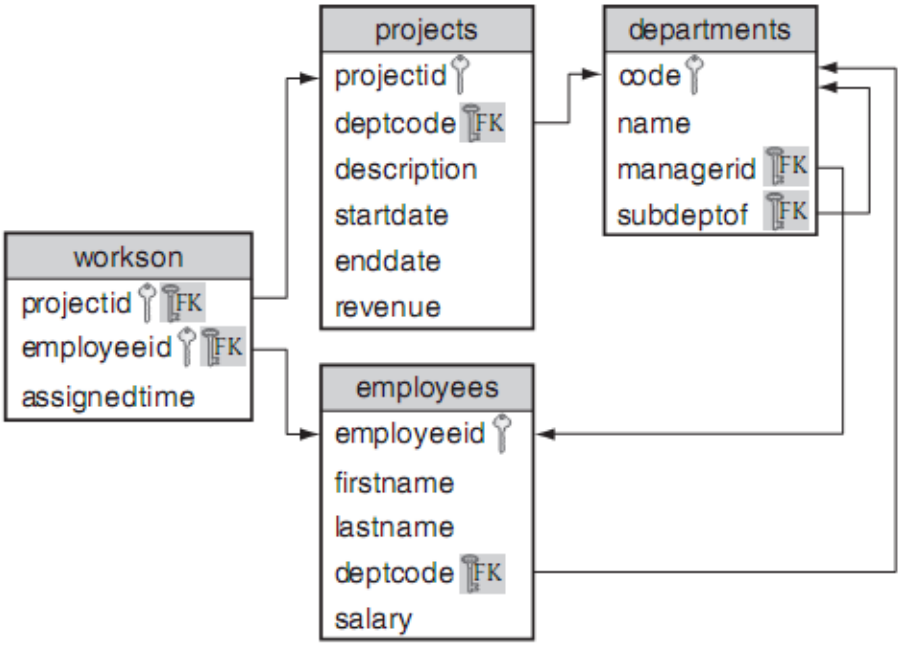
- Correlated subqueries in the SELECT clause work just like they do in the WHERE clause.

List each ingredient and its supplier.

```
SELECT name, (SELECT companyname
FROM vendors v
WHERE v.vendorid = i.vendorid) AS "supplier"
FROM ingredients i;
```

EXERCISES:

- 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.
- Not all of these queries can be solved using subqueries. Some of them require correlated-subqueries and division operation. They will be discussed in the next lab.



1. Find the names of all people who work in the Consulting department.
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.
3. Find the total percentage of time assigned to employee Abe Advice.
4. Find the names of all departments not currently assigned a project.
5. Find the first and last names of all employees who make more than the average salary of the people in the Accounting department.
6. Find the descriptions of all projects that require more than 70% of an employee's time.
7. Find the first and last name of all employees who are paid more than someone in the Accounting department.
8. Find the minimum salary of the employees who are paid more than everyone in the Accounting department.
9. Find the first and last name of the highest paid employee(s) in the Accounting department.
10. For each employee in the department with code ACCNT, find the employee ID and number of assigned hours that the employee is currently working on projects for other departments. Only report an employee if she has some current project to which she is assigned more than 50% of the time and the project is for another department. Report the results in ascending order by hours.
11. Find all departments where all of their employees are assigned to all of their projects.
12. Use correlated subqueries in the SELECT and WHERE clauses, derived tables, and subqueries in the HAVING clause to answer these queries. If they cannot be answered using that technique, explain why.
    (a) Find the names of all people who work in the Information Technology department.
    (b) Find the names of all people who work in the Information Technology department and who spend more than 20% of their time on the health project.
    (c) Find the names of all people who make more than the average salary of the people in the Accounting department.
    (d) Find the names of all projects that require more than 50% of an employee's time.
    (e) Find the total percentage time assigned to employee Bob Smith.
    (f) Find all departments not assigned a project.
    (g) Find all employees who are paid more than someone in the Information Technology department.
    (h) Find all employees who are paid more than everyone in the Information Technology department.
    (i) Find the highest paid employee in the Information Technology department.

Reference:

- This lab sheet is prepared from the book: SQL: Practical Guide for Developers. Michael J. Donahoo and Gregory D. Speegle.

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