Lecture Notes # 4

It is the aim of this script to improve skills in **algebraic query expressions** and in **SQL SELECT statement**. [1].

Introduction to SQL (Query)

(Algebraic) Query Language

Before to introduce SQL queries, we shall learn an algebra - called **relational algebra** -, that make easy to comprehend how to **operate on relations**: **a way to construct new relations from given relations.**

Is it really essential to learn the relational algebra? Although it is not currently used in commercial DBMS as a query language, **SQL query language incorporates relational algebra.**

What is an algebra? Specifically the relational algebra consists of operators and atomic operands: variables (= relations) and constants (= finite relations).

The operators of the traditional algebra fall into four classes:

1. Set operations on relations: *union*, *intersection* and *difference*.
2. *Selection*, to eliminate some tuples, and *Projection*, to eliminate some columns.
3. *Cartesian Product* pairs the tuples of two relations in all possibile ways, and *Join* selectively pairs tuples from two relations.
4. *Renaming*: simply changes the attribute names of the relation schema.

The relational algebra operates by means of an expression, referred as *query*.

In the next sections both queries are shown: the algebraic expression and the SQL statement. This surely help us to better understand how an SQL statement works.

[1] The script is mainly based on "A First Course in Database Systems", J. Ullman, J. Widom. Mostly examples and exercises are created by the author of this script.

The SELECT Statement

Query statements return a result in table form.

Distinguishing feature of SQL is to express queries declaratively in terms of relational expressions.

Syntax of SQL SELECT statement

SELECT ListOfAttributesOrExpressions
FROM ListOfTables
[WHERE ConditionsOnTuples]
[GROUP BY ListOfGroupingAttributes]
[HAVING ConditionsOnAggregates]
[ORDER BY ListOfOrderingAttributes]

The SELECT statement consists in six clauses. *Keywords* (words of the language) are uppercase letters, even SQL is insensitive to uppercase or lowercase letters. Optional clauses are in brackets.

We assume the relation "vegetable":

vegetable	
name	**colour**
onion	red
pepperoni	red
pepperoni	yellow
carrot	orange

- The SELECT clause tells which attributes of the tuples are reported as a part of the answer.
- The FROM clause gives the relation (or relations) to which the query refers;

```
SELECT name, colour
FROM vegetable
```

The * symbol indicates that the entire tuple is reported;

```
SELECT *
FROM vegetable
```

- Almost SQL queries use the WHERE clause, in order to return tuples satisfying a condition;

```
SELECT *
FROM vegetable
WHERE colour = 'red'
```

Projection and Selection in SQL

(1) In the *relational algebra*:

(a) **Selection** operator, applied to a relation R, produces a new relation with a subset of R's tuples, generally satisfying some condition C that involves the attributes of R:

$\sigma_C(R)$, where C is a conditional expression;

(b) **Projection** operator is used to produce from a relation R a new relation that has only some of R's columns:

$\pi_{A_1,A_2,..,A_n}(R)$, where $A_1, A_2, .., A_n$ are attributes of R;

(c) In order to control the names of the attributes used for relations, it could be convenient to use the operator **renaming** that renames attributes relation:

$\rho_{(A_1^*,A_2^*,..,A_3^*)}(R)$, where the attributes of the a new relation S are named $A_1^*, A_2^*, .., A_3^*$;

(2) We assume the relations "vegetable" and "nutritive value";

vegetable	
name	**colour**
onion	red
tomato	red
pepperoni	yellow
carrot	orange

nutritive value		
name	**vitamin**	**mineral salt**
pepperoni	1.2	0.8
carrot	2.0	1.2

- An SQL query can **project** the relation onto some of its attributes: in the SELECT clause are listed some of the attributes of the relation mentioned in the FROM clause;

$$[\pi_{name}(\sigma_{colour=\text{`}red\text{'}}(vegetable))]$$

```
SELECT name
FROM vegetable
WHERE colour = 'red'
```

The column header can be modified by the keyword AS and an *alias*, which becomes the new header in the resulting table;

$$[\rho_{(vegetable\_name)}(\pi_{name}(\sigma_{colour=\text{`}red\text{'}}(vegetable)))]$$

```
SELECT name AS vegetable_name
FROM vegetable
WHERE colour = 'red'
```

Relation can be projected on new attributes built trough **expressions** in the SELECT clause[2];

$$[\rho_{(nutritive)}(\pi_{vitamin+mineral\_salt}(\sigma_{name=\text{`}pepperoni\text{'}}(nutritive\_value)))]$$

```
SELECT vitam + mineral_salt AS nutritive
FROM nutritive_value
WHERE name = 'pepperoni'
```

- A **selection** of tuples from the relation is operated through the WHERE clause. The WHERE clause is followed by a condition, an expression made up by attributes, values (numeric or text), and comparison operators: =, <>, <, >, <=, and >=. The following expressions could be valid conditions in the WHERE clause;

```
1) name = 'pepperoni'
2) vitamin >= 1.2
3) vitamin <= mineral_salt
```

[2](+) mathematical addition.

Expressions can use logical operators: **AND**, **OR**, and **NOT**, and they can contains parenthesis, for instance these expressions could be valid conditions of the WHERE clause;

```
1) name = 'pepperoni' OR colour = 'red'
2) name = 'pepperoni' AND colour = 'red'
3) NOT(colour = 'red')
```

The comparison evaluates either **TRUE** or **FALSE**.

Join in SQL

(1) In the *relational algebra*:

(a) The **cartesian (cross) product** of two relations R and S is the set of pairs that can be formed by choosing the first tuple of the pair to be any tuple of R and the second any tuple of S:

$R \times S$;

(b) The **natural join** of two relations R and S pairs tuples from R and S that **agree** in whatever attributes, **common** to schemas of R and S:

$R \bowtie S$;

(c) The **theta join** of two relations R and S pairs tuples from R and S that satisfy a **condition** C:

$R \bowtie_C S$, where **for instance** C equals the attribute $R.A$ from R and the attribute $S.A$ from S ($R.A = S.A$).

(2) SQL query can combine two or more relations through joins (even through cartesian products). Joins (and cartesian products) are realized through two or more relations in the FROM clause.

(3) SQL has a simple way to couple relations in one query, that is to realise a cartesian product: list each relation in the FROM clause, then the SELECT and the WHERE clauses can refer to the attributes of any of the relations in the FROM clause. Join instead needs that relations agree in one or more attributes, as declared in the WHERE clause.

(4) **Sometimes a query involves several relations having two or more attributes with the same name. SQL solves this problem by allowing to place a relation name and a dot in front of an attribute (disambiguation).**

(5) We assume the relations *vegetable* and *nutritive value*;

vegetable	
name	**colour**
onion	red
tomato	red
pepperoni	yellow
carrot	orange

nutritive value		
name	**vitamin**	**mineral salt**
pepperoni	1.2	0.8
carrot	2.0	1.2

- *vegetable* × *nutritive value* (cartesian product)

$$[vegetable \times nutritive\_value]$$

```
SELECT *
FROM vegetable, nutritive_value
```

vegetable × nutritive value				
name	**colour**	**name**	**vitamin**	**mineral salt**
onion	red	pepperoni	1.2	0.8
onion	red	carrot	2.0	1.2
tomato	red	pepperoni	1.2	0.8
tomato	red	carrot	2.0	1.2
pepperoni	yellow	pepperoni	1.2	0.8
pepperoni	yellow	carrot	2.0	1.2
carrot	orange	pepperoni	1.2	0.8
carrot	orange	carrot	2.0	1.2

- *vegetable* **join** *nutritive value*[3]

$[\pi_{name,colour,vitamin,mineral\_salt}(\sigma_{vegetable.name=nutritive\_value.name}(vegetable \times nutritive\_value))]$

```
SELECT vegetable.name, vegetable.colour,
          nutritive_value.vitamin, nutritive_value.mineral_salt
FROM vegetable, nutritive_value
WHERE vegetable.name = nutritive_value.name
```

vegetable × nutritive value				
name	**colour**	*name*	**vitamin**	**mineral salt**
pepperoni	**yellow**	*pepperoni*	**1.2**	**0.8**
carrot	**orange**	*carrot*	**2.0**	**1.2**

An alternative join (explicit **Theta join**)

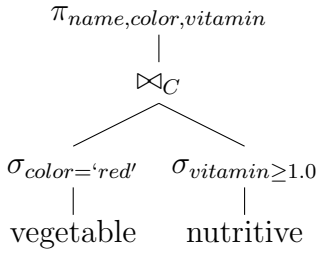
$[\pi_{name,colour,vitamin,mineral\_salt}(vegetable \bowtie_{vegetable.name=nutritive\_value.name} nutritive\_value)]$

From Queries

Relational algebra allows to form expressions of arbitrary complexity by applying operations to the result of other operations. It is helpful to represent expressions as trees.

Suppose we want to know "What are vegetables **red coloured** having **at least 1.0** of vitamin?"

1. Select those `vegetable` tuples that have `colour` equal to `red`.
2. Select those `nutritive` tuples that have `vitamin` greater than or equal to `1.0`.
3. Compute the Theta-Join (`C: vegetable.name = nutritive_value.name`).
4. Project the resulting relation onto attributes `name`, `colour`, `vitamin`.



There is often more than one relational algebra expression that represents the same computation.

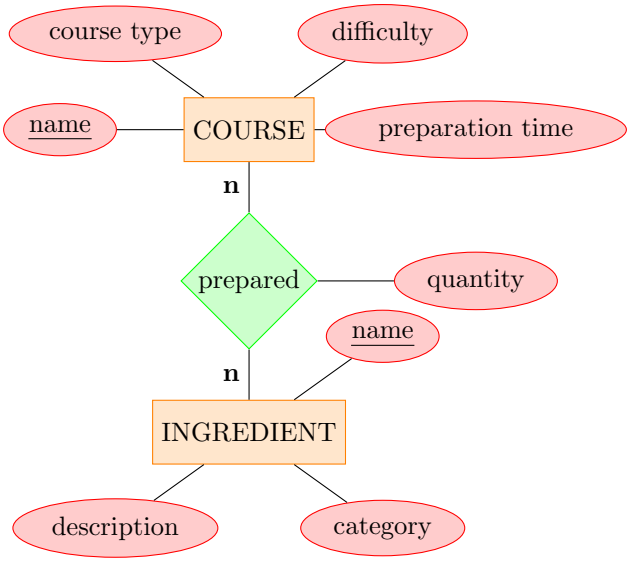
[3]The attribute *name*, in italic, is not projected by the select statement, but it is referred in order to show the appropriate tuples.

All database systems have a query-answering system, and many of them are based on a language that is similar in expressive power to relational algebra. Thus, the query asked by a user may have many equivalent expressions. The query ***optimizer*** *replaces one expression of relational algebra by an equivalent expression that is more efficiently evaluated.*

Practice Session: The SELECT Statement

Let assume the the database "Book - Recipe" [Script: `recipe.sql`].

E/R Diagram



Database Schema

Course(name, courseType, difficulty, preparationTime)

Ingredient(name, description, category)

Recipe(Course.courseName, Ingredient.ingredientName, quantity)

Query

Write SQL query in order to answer the following questions:

1. List course in **book-recipe** database. Report only course name and the corresponding type.

```
SELECT name, courseType
FROM Course;
```

2. List ingredients courses are made up. Report all available data.

```
SELECT *
FROM Ingredient;
```

3. List ingredients of the category "spice".

```
SELECT *
FROM Ingredient
WHERE category = 'spice';
```

4. List ingredients of the category "vegetable". Report only names, and instead of the header "name" display the header "vegetable".

```
SELECT name AS vegetable
FROM Ingredient
WHERE category = 'vegetable';
```

5. Show printable labels describing book courses. Use this 'format': [4]

   `COURSE: Lasagne (PASTA)`

```
SELECT CONCAT('COURSE: ', name, ' (', UCASE(courseType), ')') AS label
FROM Course;
```

6. List all courses, excluding those of type "dessert". Report only name, type, and difficulty;

```
SELECT name, courseType, difficulty
FROM Course
WHERE NOT(courseType = 'dessert');
```

7. List all courses, excluding those of type "dessert" and further excluding those with more than 50 minutes preparation time. Report only name, type, and difficulty.

```
SELECT name, courseType, difficulty
FROM Course
WHERE NOT(courseType = 'dessert') AND (preparationTime <= 50);
```

[4] The function CONCAT() is available and the equivalent operator "||", but this one only if `sql_mode = 'PIPES_AS_ACCOUNT'` is set.

8. List courses which requires at least 50 minutes and no more than 100 minutes to prepare.

```
SELECT *
FROM Course
WHERE (preparationTime >= 50 AND preparationTime <= 100);
```

9. List ingredients different from 'vegetables' and 'spices'.

```
SELECT *
FROM Ingredient
WHERE NOT(category = 'vegetable' OR category = 'spice');
```

10. Basically ingredients are used to prepare a course, according to the recipe. Report ingredient name, with the corresponding category, and in which recipe the ingredient participates.

```
SELECT ingredient.name, ingredient.category, recipe.courseName
FROM Ingredient, Recipe;
```

11. The returned table of the previous query looks so strange! All Ingredients just paired with all courses it does not make sense!

    It is clear that we have to relate ingredients to the course they are required.

```
SELECT ingredient.name, ingredient.category, recipe.courseName
FROM Ingredient, Recipe
WHERE Ingredient.name = Recipe.ingredientName;
```

12. Now, a typical question could be: *Which course requires eggs?*;

```
SELECT ingredient.name, ingredient.category, recipe.courseName
FROM Ingredient, Recipe
WHERE Ingredient.name = Recipe.ingredientName
AND Ingredient.name = 'Egg';
```

13. We want to know how often **Olive Oil** is used as ingredient in the book-recipe courses. Report ingredient name, course type, course name, and the corresponding quantity of **Olive Oil** required. Display the quantity adding the label (concatenate) `Cl`.

```
SELECT ingredientName, courseType, courseName, CONCAT(quantity, ' Cl') AS qty
FROM Recipe, Course
WHERE Recipe.courseName = Course.name
AND ingredientName='Olive Oil';
```

14. We aim to list ingredients belonging to *fruit* and *vegetable* **category**. Report the ingredient category, the ingredient name and the course in which it is used.

```
SELECT Ingredient.category, Ingredient.name, Recipe.courseName
  FROM Recipe, Ingredient
 WHERE Recipe.ingredientName = Ingredient.name
   AND (Ingredient.category = 'fruit' OR Ingredient.category = 'vegetable');
```

15. Everybody likes pasta! But someone could suffer from food intolerance! Show courses of course type **pasta** containing **eggs**.

```
SELECT course.name
  FROM Course, Recipe
 WHERE course.name = recipe.CourseName
   AND course.courseType = 'Pasta'
   AND recipe.ingredientName = 'Egg';
```

16. You would like something sweet and fat! Your favorite courses should have no more than 200 grams of white sugar or alternatively no more than 400 grams of mascarpone.

```
SELECT *
  FROM recipe
 WHERE (ingredientName = 'White sugar' AND quantity <= 200)
    OR (ingredientName = 'Mascarpone' AND quantity <= 400);
```

17. Are there available at least two distinct courses prepared with the same ingredient?

```
SELECT R1.courseName, R1.ingredientName, R2.courseName
  FROM Recipe R1, Recipe R2
 WHERE R1.ingredientName = R2.ingredientName
   AND R1.courseName <> R2.courseName;
```

18. Detect something "easy" to prepare, containing at least one vegetable like: carrot, celery, Lettuce.

```
SELECT courseName, ingredientName, preparationTime
  FROM course, recipe
 WHERE course.name = recipe.CourseName
   AND ingredientName IN ('carrot','celery','Lettuce')
   AND difficulty = 'easy';
```

Grouping and Aggregation

Grouping-and-Aggregation operator allows us to **partition** the tuples of a relation into **groups**, based on the values of tuples in one or more attributes.

1. **Grouping**: to group tuples, we use a GROUP BY clause followed by a list of grouping attributes. Whatever aggregation operators are used, in the SELECT clause, operators are applied only within groups.
2. **Aggregation**: SQL uses the five **aggregation operators** : SUM, AVG (average), MIN, MAX, COUNT. These operators are used by applying them to a scalar-valued expression (an attribute), in a SELECT clause.

Aggregated operator is applied to an **attribute** or **expression** involving attributes, and are evaluated on a per-group basis.

Remark: SELECT clause that has aggregations, **only those attributes that are mentioned** in the GROUP BY clause **may appear un-aggregated** in the SELECT clause.

The operator COUNT has two expressions:

- COUNT(*) or COUNT(*attribute or expression*), which counts all the tuples in the relation;
- COUNT(DISTINCT *attribute*), which count the number of distinct values in *attribute*;
- Example: *How many courses are registered in the database?*

```
SELECT COUNT(*)
  FROM Course;


SELECT COUNT(name)
  FROM Course;
```

- Example: *Determine and then report for each* ***difficulty*** *level, the number of courses so classified, and for this level the highest preparation time and the lowest preparation time.*

```
SELECT difficulty, COUNT(name) AS nr_course,
       MIN(preparationTime) AS min_time, MAX(preparationTime) AS max_time
  FROM course
GROUP BY difficulty;
```

$$\gamma_{difficulty, COUNT(name), MIN(preparationTime), MAX(preparationTime)}(course)$$

a) For each ingredient category compute and show the number of ingredients required for courses. **Consider only cheese and vegetable categories.**

```
SELECT category, COUNT(name) AS nr_ingr
  FROM ingredient
 WHERE category IN ('cheese', 'vegetable')
GROUP BY category;
```

b) Butter or Olive Oil? Which one is mostly used in cooking? Compute the number of courses in which Butter and Olive Oil are used.

```
SELECT ingredientName, COUNT(*) AS nr
FROM recipe
WHERE ingredientName IN ('Butter', 'Olive oil')
GROUP BY ingredientName;
```