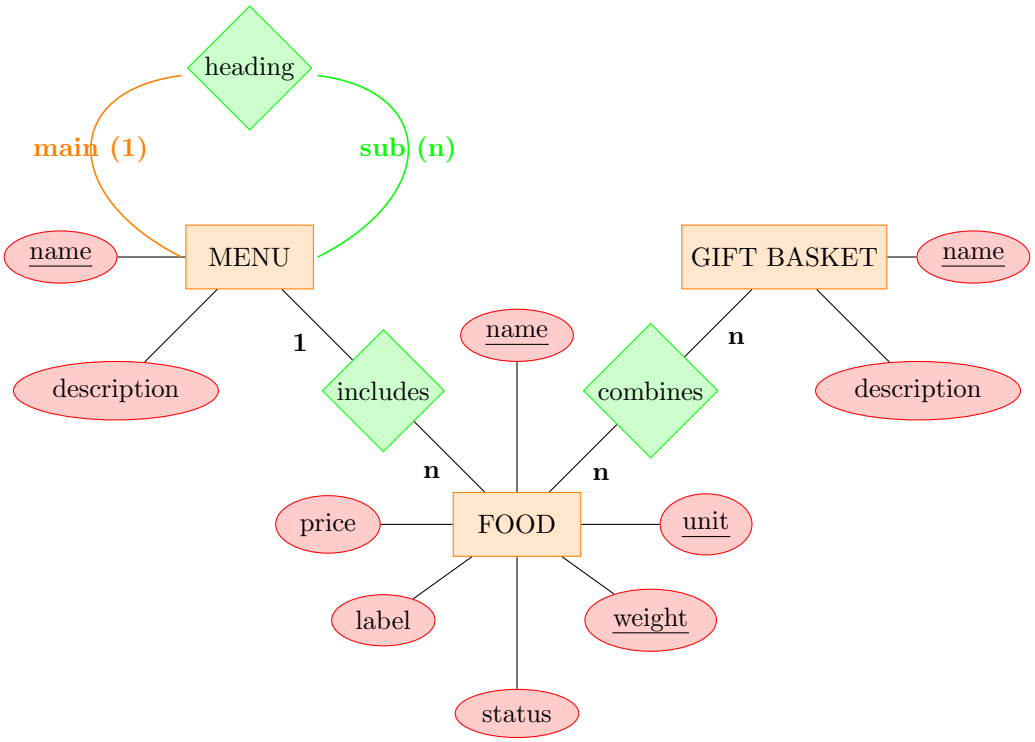
Lecture Notes # 5

Advanced topics on SELECT Statement

Let assume the the database "Online Market" [Script: `online_market_small.sql`], which stores **only** "basic information" for the online shop.

E/R Diagram



Database Schema

Menu(name, description, main)

Food(name, weight, unit, label, price, status, Menu.menu_name)

GiftBasket(name, description)

BasketCombines(GiftBasket.name, Food.name, Food.weight, Food.unit)

Let assume the SELECT statement syntax.

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

Basic query

1. Someone says that the gift basket *Savory Basket* is very gourmand. Show all food products included in the basket and the corresponding prices.

```
SELECT basketCombines.basket_name, Food.name, Food.price
  FROM BasketCombines, Food
 WHERE BasketCombines.food_name = Food.name
   AND BasketCombines.food_weight = Food.weight
   AND BasketCombines.food_unit = Food.unit
   AND BasketCombines.basket_name = 'Savory basket';
```

2. Pistachio is your favorite ingredient! Show food name, weight,unit, label and price of *Pistachio Cream*.

```
SELECT name, weight, unit, label, price
  FROM Food
 WHERE name = 'Pistachio Cream';
```

3. Main menus are regular menus grouping many menu items. Verify this fact!

```
SELECT a.name, a.main
  FROM Menu a, Menu b
 WHERE a.name = b.main;
```

[**Notice:** Observe "main" is NULL.]

Sub-query

A SELECT statement can be a part of another SELECT statement, this is called a *sub-query*:

- A sub-query can return a single constant (number, string, date), that is **compared** with a value in the WHERE clause of the main SELECT statement;

  Example: *Return the food product in the shop, which is the most expensive.*

```
SELECT name, weight, unit, price
  FROM Food
 WHERE price = (SELECT MAX(price)
                  FROM Food);
```

- A sub-query can appear in FROM clause, just like any **relation (table)**;

  Example: *List food products not available on the online-shop anymore.*

  Firstly create a query in order to select foods currently not available.

```
SELECT new.name, new.weight, new.unit
    FROM (SELECT *
            FROM Food
           WHERE status = true) new;
```

Operators

There are some SQL operators that can be applied to a relation **R** **that usually is a result of a sub-query** and produce a boolean result in the WHERE clause. Follows some helpful operators:

- **EXISTS R** is a condition that is true if and only if **R** is not empty. Likewise, **NOT EXISTS R** is true if and only if **R** is empty.

  Example: *List food products that are* **at least** *in one gift basket.*

```
SELECT name, weight, unit, price
  FROM Food
 WHERE EXISTS (SELECT *
                 FROM BasketCombines
                WHERE BasketCombines.food_name = Food.name
                  AND BasketCombines.food_weight = Food.weight
                  AND BasketCombines.food_unit = Food.unit);
```

The sub-query shall be evaluated many times, once (at least) for each value that comes from a tuple of the outer query, this sub-query is called a **correlated sub-query**.

[**Notice**: Observe, simply join the tables `basketCombines` and `food` we have a different list of tuples!]

- s **IN** R is true if and only if s is **equal to one of the values** in R. Likewise, s **NOT IN** R is true if and only if s is equal to no value in R. R **has only one attribute**, hence R is equal to **a set of values**.

  Example: *Show menus listing food products weighted in "grams".*

```
SELECT main, name
  FROM Menu
 WHERE name IN (SELECT menu_name
                  FROM Food
                 WHERE unit = 'g')
 ORDER BY main;
```

- $s >, [<, =]$ **ALL** R is true if and only if s is **greater than every value** in R, this one **having only one attribute**, hence R is equal to a set of values. **NOT** $s >, [<, =]$ **ALL** R is true if and only if s is not the maximum value in R.

  Example: *Which food products are more expensive than all food no more available?*

```
SELECT *
  FROM Food
 WHERE status = true
   AND price >ALL (SELECT price
                     FROM Food
                    WHERE status = false);
```

- $s >, [<, =]$ **ANY** R is true if and only if s is **grater than at least one value** in R, this one **having only one attribute**, hence R is equal to a set of values. **NOT** $s >$ **ANY** R is true if and only if s is the minimum value in R.

Example: *Which food has been added with a different quantity: for example larger weight?*

```
SELECT *
  FROM Food a
 WHERE weight >ANY (SELECT weight
                      FROM Food b
                     WHERE b.name = a.name);
```

A correlated subquery is used in order to combine tuples of the same relation R.

Join Expressions

In this section are presented expressions that can be used to join two or more relations[1].

- Generally we couple relations in a simple way: list each relation in the FROM clause. Then the WHERE clause can refer to the attributes that relates the relations;

```
SELECT Menu.name, Menu.main, Food.name, Food.weight, Food.unit
  FROM Food, Menu
 WHERE Food.menu_name = Menu.name;
```

```
SELECT basket_name, Food.name, Food.weight, Food.unit, Food.price
  FROM BasketCombines, Food
 WHERE BasketCombines.food_name = Food.name
   AND BasketCombines.food_weight = Food.weight
   AND BasketCombines.food_unit = Food.unit;
```

- **Explicit Join**: join can be obtained with the keywords **JOIN** and **ON** in the **FROM** clause;

```
SELECT Menu.name, Menu.main, Food.name, Food.weight, Food.unit
  FROM Food JOIN Nenu ON Food.menu_name = Menu.name;
```

[1]Join expressions in SQL standard could not be supported by some DBMS vendors, or it could have different syntaxes.

- **Natural Join**: the join condition is that all pairs of attributes from the two relations having a **common name (*)** are equated, and there are no other conditions. ***One*** *of each pair of equated attributes are projected out.* Natural join can be obtained with the keywords NATURAL JOIN in the FROM clause.

```
SELECT *
  FROM (SELECT name AS menu_name, main # (*)
          FROM Menu) m NATURAL JOIN Food;
```

$$\rho_{menu\_name}(menu) \bowtie food$$

- **Outer Join**: it pads dangling tuples from both of its relation arguments. It is denoted as *full* outer join.

```
SELECT *
  FROM Food FULL OUTER JOIN BasketCombines
    ON (Food.name = BasketCombines.food_name
        AND Food.weight = BasketCombines.food_weight
        AND Food.unit = BasketCombines.food_unit);
```

$$food \bowtie^{Full} basketCombines$$

Left and **Right** outer join pad dangling tuple respectively from the left and from the right.

```
SELECT *
  FROM Food LEFT OUTER JOIN BasketCombines
    ON (Food.name = BasketCombines.food_name
        AND Food.weight = BasketCombines.food_weight
        AND Food.unit = BasketCombines.food_unit);
```

$$food \bowtie^{Left} basketCombines$$

Union, Intersection and Difference

Sometimes we need to combine relations using the well known set operations, that in *Relational Algebra* are:

- The **union** of R and S is the set of elements that are in R or S or both;
- The **intersection** of R and S is the set of elements that are in both R and S;
- The **difference** of R and S is the set of elements that are in R but not in S.

SQL provides corresponding operators, that produce relations with the same list of attributes and attribute types[2]:

(a) *union*: `UNION [ALL]`

```
(SELECT name, weight, unit, price, 'cheaper' AS target
   FROM Food
  WHERE price <= 5.0)
UNION
(SELECT name, weight, unit, price, 'expensive' AS target
   FROM Food
  WHERE price >= 15.0);
```

$$food_{price \leq 5} \cup food_{price \geq 15}$$

(b) *intersection*: `INTERSECT`

```
(SELECT name, weight, unit, price, 'regular' AS target
   FROM Food
  WHERE price <= 15.0)
INTERSECT
(SELECT name, weight, unit, price, 'expensive' AS target
   FROM Food
  WHERE price >= 10.0);
```

$$food_{price \leq 15} \cap food_{price \geq 10}$$

[2]Set operators in SQL standard could not be supported by some DBMS vendors, or it could have different syntaxes.

(c) *difference*: EXCEPT

```
(SELECT name, weight, unit, price, 'regular' AS target
  FROM Food
 WHERE price <= 15.0)
EXCEPT
(SELECT name, weight, unit, price, 'expensive' AS target
  FROM Food
 WHERE price >= 10.0);
```

$$food_{price \leq 15} - food_{price \geq 10}$$

Eliminating duplicates

A relation, being a set, can not have more than one copy of any given tuple, while a SELECT statement creates a new relation that does not ordinarily eliminate duplicates, thus the SELECT response may list the same tuple several times.

If we do not wish duplicates in the result, then we may follow the keyword SELECT by the keyword **DISTINCT**.

Example: *Show (only once) main menu.*

```
SELECT DISTINCT(main)
  FROM Menu;
```

Grouping and Aggregation: Choose Groups

Sometimes we want to choose groups based on some aggregate property of the group itself. Then we follow the GROUP BY clause with a **HAVING** clause, followed by a condition about the group.

Example: *For each menu determine the cheapest food product. Show menu which offers at least one product cheaper than 5 €.*

```
   SELECT menu_name, MIN(price) AS cheaper
     FROM Food
GROUP BY menu_name
  HAVING MIN(price) <= 5;
```

$$\sigma_{MIN(price)<=5}(\pi_{menu\_name,MIN(price)}(\gamma_{menu\_name}(food)))$$

Comparison of String

Two strings are equal if they are the same sequence of characters. SQL provides the capabilities to compare strings on the basis of a **pattern match** s `LIKE` p where s is a string and p is a **pattern**.

The pattern is a string with the optional use of the two special characters '%' and '_':

- Ordinary characters in p match only themselves in s;
- '%' in p can match any sequence of zero or more characters in s;
- '_' in p matches any one character in s;

The value of this expression is true if and only if string s matches pattern p.

Example: *List 'organic' food products*;

```
SELECT menu_name, name, weight, unit
  FROM Food
 WHERE name LIKE '%organic%';
```

The value that the column 'name' holds is the string s, while '%organic%' is the pattern p.

Example: *Pistachio is your favorite ingredient! Show food name, unit weight, label and price of the cheapest one*;

```
SELECT name, weight, unit, label, price
  FROM Food
 WHERE name LIKE '%pistachio%'
   AND price = (SELECT MIN(price)
                  FROM Food
                 WHERE name LIKE '%pistachio%');
```

Example: *We know that cream food is appreciated very much, but are we offering enough cream food products in the menus? Compute the number of cream food products for each menu.*

```
SELECT menu_name, COUNT(*) AS nr
  FROM Food
 WHERE menu_name LIKE '%cream%'
GROUP BY menu_name
ORDER BY COUNT(*) DESC;
```

Date and Time

SQL supports dates and times as special data type (SQL standard notation is very specific about format).

1. A date constant is represented by the keyword `DATE` followed by a quoted string of a special form, for example `DATE '2023-05-01'`;
2. A time constant is represented similarly by the keyword `TIME` and a quoted string, for example `TIME '14:00:10'`;
3. SQL provides few **functions** to manage date and time.

Example: *List food products no more available, label these products with the last date availability* ***31/12/2022****.*

```
SELECT menu_name, name, unit, weight, '2022-12-31' AS last_date
  FROM Food
 WHERE status = false;
```

NULL Values

SQL allows attributes to have a special value, called the `NULL` value, that can have different interpretations as for instance *value unknown*, *value inapplicable*, and *value withheld*.

- The only way to check if *attribute* has the value `NULL` is with the expression *attribute* `IS NULL`;
- This expression has the value TRUE if *attribute* has the value `NULL` and it has a value FALSE otherwise;
- Similarly, *attribute* `IS NOT NULL` has the value TRUE unless the value of *attribute* is `NULL`;

Example: *List products, with which a declared availability is not specified.*

```
SELECT *
  FROM Food
 WHERE status IS NULL;
```

Ordering

The tuples produced by a query can be presented in sorted order. The order may be based on the value of any attribute. To get output in sorted order, the ORDER BY clause is added.

The order is be ascending `ASC` (default) or descending `DESC`.

Example: *List all food products from the most expensive to the cheapest one.*

```
SELECT menu_name, name, weight, unit, price
  FROM Food
 ORDER BY price DESC;
```

Advanced Queries

It is the aim of this section to improve query skills, therefore we move to the full database design, which includes track of user's browsing and a greater sample of foods.

[`online-market-large.sql`]

a) We have the need to verify whether in the main-menu **Preserved Foods** there is at least one food having a price larger than 8.50 €. Show the sub-menu they belong to.

```
SELECT name, main
  FROM Menu
 WHERE main = 'Preserved Foods'
   AND EXISTS (SELECT *
                 FROM Food
                WHERE price > 8.50
                  AND Food.menu_name = Menu.name);
```

b) Create a short **price list** including foods of the two sub-menu **Sauces and Pesto** and **Olives and Capers**. For the first sub-menu a special discount of 20% is applied. Report the food name, weight and unit, the price, compute the discounted price and beside a special column with the label **'discounted'** if the price is discounted the label **'full'** if not.

[**Tip:** Select data for each sub-menu separately. For the full prices do not compute anything and just return the `NULL` value.]

```
(SELECT name, weight, unit, price, price*0.80 AS n_price, 'discounted' AS type
   FROM Food
  WHERE menu_name = 'Sauces and Pesto')
UNION
(SELECT name, weight, unit, price, NULL AS n_price, 'full' AS type
   FROM Food
  WHERE menu_name = 'Olives and Capers')
ORDER BY weight;
```

c) We create for each product a sort of "tag" following this text format:

`Traditional Bolognese Ragù - [La Dispensa di Amerigo:  180.00 g]`

which includes name, label, weight, unit. Other than the label the query must return **NULL** if the food does not belong to any gift basket, the gift basket name (capitalized) otherwise. [**Tip:** the explicit join could be helpful]

```
SELECT CONCAT(name, ' - [',label, ': ', weight, ' ', unit, ']') AS label,
       UCASE(basket_name) AS basket
  FROM Food LEFT JOIN BasketCombines
    ON (Food.name = BasketCombines.food_name
        AND Food.weight = BasketCombines.food_weight
        AND Food.unit = BasketCombines.food_unit)
ORDER by name DESC;
```

d) We aim to detect main menus that suffer of a poor offer of food products. Return **main menu** which offer less than **seven** food products, additional data is the available number of food products in this menu and their commercial value, that is the total price of the food products.

```
SELECT main, COUNT(*) AS nr, SUM(price) AS value
  FROM Food, Menu
 WHERE Food.menu_name = Menu.name
   AND status = true
GROUP BY main
HAVING nr < 7;
```

e) Realize a comparative selection of your food products. The selection is based on **cost/benefit** parameter and on the level of **nutritive** we can benefit, specifically *cost/benefit parameter* equals the ratio price/weight, while *nutritive benefit* is the 30% of the weight. Show those foods which have the *cost/benefit parameter* larger than its average (over all foods) **or** the *nutritive benefit* larger than its average (over all foods).

```
SELECT name, price/weight AS comparative, weight*0.30 AS nutritive
  FROM Food
 WHERE price/weight > (SELECT AVG(price/weight)
                       FROM Food)
    OR weight*0.30 > (SELECT AVG(weight*0.30)
                      FROM Food);
```

f) The goal is to verify if the food products proposal can be appreciated by different customer segment, hence we compute two different key numbers. The first one is for each main menu the number of foods having a price smaller than or equal to 5 € (adding the label 'cheaper') while the second one, again for each main menu, the number of foods having a price larger than or equal to 14 € (adding the label 'expensive').

Both results must be returned together in the same list.

```
(SELECT 'cheaper' AS type, Menu.main, COUNT(*) AS nr
   FROM Food, Menu
  WHERE Food.menu_name = Menu.name
    AND price <= 5.0
GROUP BY Menu.main)
UNION
(SELECT 'expensive' AS type, Menu.main, COUNT(*) AS nr
   FROM Food, Menu
  WHERE Food.menu_name = Menu.name
    AND price >= 14.0
GROUP BY Menu.main)
```

g) You aim to offer a "Smart Basket", made by a selection of cheapest products. You identify among your food products those that in the corresponding menu they are the cheapest ones.

Show returned foods from the most expensive to the cheapest one.

[**Tip**: the operator IN works with a couple of attribute: `WHERE (attr1, attr2) IN`.]

```
SELECT menu_name, CONCAT(name,'-',weight,'-', unit), price
  FROM Food
 WHERE (menu_name,price) IN (SELECT menu_name, MIN(price)
                               FROM Food
                           GROUP BY menu_name)
ORDER BY price DESC;
```

h) We have a selection of **Preserved Foods** in the online market which could be appreciated in the United Kingdom market, hence for each food product in the main menu **Preserved Foods** we create a label (specifically **uk_label**) concatenating the name, the weight and the price, paying attention to UK units (ounces=oz for grams, pound=£ for euro). [**Tip**: In order to convert values in ounces and pounds multiply respectively the values by 0.035 and 0.8.] One acceptable label could be:

Black Olive Paste: 2.80 oz # 2.160 £

```
SELECT concat(Food.name, ': ', round(weight*0.035,2),
                        ' oz # ', price*0.8, ' £') AS uk_label
  FROM Menu, Food
 WHERE Menu.name = Food.menu_name
   AND main = 'Preserved Foods';
```

i) Marketing manger would like to rename gift baskets emphasizing products "value", therefore he would like a report, as below. [**Tip:** apply the strategy to find maximum and minimum separately]

Basket Name	Price	Value
Delight taste	4.60	MIN
Delight taste	8.98	MAX
...	...	...

Use the explicit JOIN whenever it is necessary to join tables.

```
(SELECT B.basket_name, MAX(F.price) AS price, 'MAX' AS value
   FROM Food F JOIN BasketCombines B
     ON (F.name = B.food_name
    AND F.weight = B.food_weight
    AND F.unit = B.food_unit)
GROUP BY B.basket_name)
UNION
(SELECT B.basket_name, MIN(F.price) AS price, 'MIN' AS value
   FROM Food F JOIN BasketCombines B
     ON (F.name = B.food_name
    AND F.weight = B.food_weight
    AND F.unit = B.food_unit)
GROUP BY B.basket_name)
ORDER BY basket_name, price;
```

l) Report producers who make "chocolates". Assume a producer is a "chocolatier" if in description (producer, food sheet relations) is written the word **cioccolato**. Show for each "chocolatier" the number of chocolate foods produced.

```
SELECT producer.name, count(*) AS 'chocolatier'
  FROM Sheet, Producer
 WHERE Sheet.producer_name = Producer.name
   AND (Sheet.description LIKE '%cioccolato%'
    OR Producer.description LIKE '%cioccolato%')
GROUP BY producer.name;
```

m) Marketing aims to trade Italian food in the US market. Report food name, weight and price. We have two constraints: (1) Foods must have a value - price - in the range 6.5, 8.5 € (2) Weight must be expressed in local unit and price must be expressed in dollar according to:

Local	**US**	**Rate**
ml	cup	0.00422675
g	oz	0.035274
kg	oz	35.273
€	$	1.08

```
(SELECT name, weight*0.00422675 AS 'cup', price*1.08 AS '$'
   FROM Food
  WHERE unit = 'ml'
    AND (price >= 6.50 AND price <= 8.50))
UNION
(SELECT name, weight*0.035274 AS 'oz', price*1.08 AS '$'
   FROM Food
  WHERE unit = 'g'
    AND (price >= 6.50 AND price <= 8.50))
UNION
(SELECT name, weight*35.273 AS 'oz', price*1.08 AS '$'
   FROM Food
  WHERE unit = 'Kg'
    AND (price >= 6.50 AND price <= 8.50))
ORDER BY name;
```

Data View

A view is a **virtual table**, created by means of SQL statement, whose instance is derived from other tables by a query.

CREATE VIEW ViewName [(AttributeList)]AS **SQLSelect**

VIEW statement syntax

Views instances (or parts of them) are only calculated when they are used (open). Views can be tables in other SELECT statements.

```
CREATE VIEW Recipe.cooking (category, ingredient, frequency) AS
    SELECT Ingredient.category, Ingredient.name, COUNT(*)
      FROM Ingredient JOIN Recipe ON Ingredient.name = Recipe.ingredientName
  GROUP BY Ingredient.category, Ingredient.name
  ORDER BY Ingredient.category, Ingredient.name;
```

Typically view data are exported in order to analyze them using functionality of specialized software. The resulting export is generally a `.csv` text file.

```
category,ingredient,frequency
"baking powder",Powder,1
bread,Bread,1
bread,"White bread",1
cakes,"Sponge finger",1
cheese,Parmesan,1
cheese,Pecorino,1
coffee,Coffee,1
egg,Egg,5
flour,Wheat,3
fruit,Apple,1
meat,Bacon,1
meat,Beef,1
meat,"Bovine meat",1
"milk products",Butter,2
"milk products",Mascarpone,1
"milk products",Milk,2
oil,"Olive oil",3
pasta,"Puff pastry",1
souce,Mayonnaiese,1
spice,Nutmeg,1
spice,Pepper,1
spice,Salt,2
sugar,Sugar,3
vegetable,Carrot,1
vegetable,Celery,1
vegetable,Lettuce,1
vegetable,Onion,1
vegetable,Pumpkin,1
vegetable,Sage,1
vegetable,"Tomato puree",1
```

Normalization

Basic concepts

Country	Year	Target	Agricultural Area Thousand ha	Production Thousand ha
Italy	2013	consumer	12426	95
Italy	2013	industry	12426	95
Italy	2014	consumer	12729	103
Italy	2014	industry	12729	103
Italy	2014	export	12729	103

Table 1: Tomato production data set - **bad design**

(a) **Functional Dependency**: A Functional Dependency (FD) on a relation R (table) is a statement of the form "If two tuples (rows of the table) of R agree on attributes $A_1A_2...A_n$ (i.e. the **tuples have the same values** in their respective values for each of these attributes), then they must also agree (**they have the same value**) on another attribute B.

Formally $A_1A_2...A_n \rightarrow B$ stands for that $A_1A_2...A_n$ **functionally determine** B". [2]

Assume the Relation:

`Tomato(country, year, target, agriarea, production)`

FD: `country, year` $\rightarrow$ `agriarea, production`

Informally, FD says that if two tuples have the same value in the `country` components, and they also have the same value in the `year` components, then those two tuples must have the same value in their `agriarea` components, and the same value in their `production` components (that is in "Tomato production").

Trivial **FD**: `country, year` $\rightarrow$ `country`

The statement `country, year` $\rightarrow$ `target` is not a functional dependency, indeed it is possible that there is more than one target.

Only attributes `country, year, target` form a key for `Tomato` Relation.

(b) **Superkey**: A set of attributes that contains a key is called a **superkey**, thus every key is a super key. However, some superkeys are not (minimal) keys. **Note that every superkey functionally determines all other attributes of the relation.**

`country, year, target, agriarea` is a superkey.

Exercise

1. Given the relation `User(userID, email, password)`, these assertions are **Functional Dependencies**:

   (1) *email* $\rightarrow$ *user_ID*, *password*

   (2) *email*, *password* $\rightarrow$ *user_ID*

   Follow an example of instances, with a chronological order of tuples insertion.

user				
#	**user ID**	**email**	**password**	**functional dependency**
1	mrossi2017	mario.rossi@gmail.com	rossi15	(1) and (2) TRUE
2	m.rossi.2017	mrossi@libero.it	mario1990	(1) and (2) TRUE
3	maros	maros@hotmail.com	mr1990?!	(1) and (2) TRUE
4	marossi	mario.rossi@gmail.com	rossi15	(1) and (2) FALSE

2. Given the relation `User(userID, email, mail-provider)`, these assertions are **Functional Transitive Dependencies**:

   (1) *user_ID* $\rightarrow$ *email*

   (2) *email* $\rightarrow$ *mail-provider*

Anomalies

In a relational schema anomalies can occur, especially updating or deleting tuples.

a) **Update anomaly**: one occurrence of a fact is changed, but not all occurrences;

b) **Deletion anomaly**: a valid fact is lost when a tuple is deleted.

Consider the **relation** "recipe"

`Recipe(ingredient, category, quantity, course, type)`

and the **instance**.

ingredient	category	quantity	course	type
butter	milk product	120	apple cake	dessert
milk	milk product	20	apple cake	dessert
carrot	vegetable	50	lasagne	pasta
butter	milk product	40	lasagne	pasta
carrot	vegetable	80	lentil Soup	pasta
butter	milk product	70	mushroom risotto	pasta

Table 2: Example of bad design

Data are redundant, few of them can be figured out by using **Functional Dependencies**:

1) `ingredient` $\rightarrow$ `category`

2) `course` $\rightarrow$ `type`

3) `ingredient, course` $\rightarrow$ `quantity`

The following anomalies can occur:

(a) **Update anomaly**: if butter category value is wrong, we need to change each of its tuples! (three in this instance)

(b) **Deletion anomaly**: If carrot is not available anymore and we remove it, we lose track of the fact that **lentil soup** is of type **pasta**!

one more example ...

Consider the relation "Agrifood" and the **instance**

Country	**Surface** Km^2	**Year** Thousand ha	**Agricultural Area** Thousand t	**Olives for oil** Thousand ha	**Tomatoes** 1000 hl	**Wine**
Italy	301338	2013	12426	2852.7	95.19	41074
Italy	301338	2014	12720.15	1903.43	103.11	42705
Italy	301338	2015	12660.89	1944.79	107.18	46734
Germany	357386	2013	16699.6	0	0.33	9102
Germany	357386	2014	16724.8	0	0.33	8493
Germany	357386	2015	16730.7	0	0.33	9294
UK	242495	2013	17259	0	0	8
UK	242495	2014	17240	0	0.2	33
UK	242495	2015	17147	0	0.23	27

Again, data are redundant, few of them can be figured out by using **Functional Dependencies**:

1) `country` $\rightarrow$ `surface`

2) `country, year` $\rightarrow$ `agriarea, olives, tomatoes, wine`

(a) **Update anomaly**: if Germany surface is inexact, we need to change each of its tuples! (**three in this instance**)

(b) **Deletion anomaly**: If a country does not produce neither olives, tomatoes, wine, we lose track of its surface!

Boyce-Codd Normal Form

Definition: A relation R is in Boyce-Codd Normal Form (BCNF) if and only if whenever there is a nontrivial Functional Dependency $A_1A_2...A_n \rightarrow B$ for R, it is the case that $A_1A_2...A_n$ is a superkey for R.

- Nontrivial means B is not a member of set $A_1A_2...A_n$;
- A superkey is any superset of a key (not necessarily a strict superset);
- **Thus, an equivalent statement of the BCNF condition is that the left side of every nontrivial Functional Dependencies must contain a key.**
- We decompose relation schemas in BCNF.

Example

(a) `Recipe(ingredient, category, quantity, course, type)`

`course` → `type`

`ingredient` → `category`

`ingredient, course` → `quantity`

1. Key is {`ingredient, course`};
2. `ingredient, course` → `quantity` does not violate BCNF;
3. The left side of `course` → `type` is not a superkey ⇒ is not in BCNF;

(b) `Agrifood(country, surface, year, agriarea, olives, tomatoes, wine)`

`country, year` → `agriarea, olives, tomatoes, wine`

`country` → `surface`

1. Key is {`country, year`};
2. `country` → `surface`, the left side is not a superkey, then is not in BCNF.

Decomposition into BCNF

- **Given** a relation R with F = $\{F_1, F_2, ...\}$, a set of Functional Dependencies.
- The **goal** is to decompose R into relations $R_1, ..., R_m$ such that:
  1. Each R_i is a projection of R, with respect to the projection of Functional Dependencies in F;
  2. Each R_i is in BCNF;
  3. **R results as the natural join of $R_1, ..., R_m$**.
- The **intuition** is to broke R into pieces:
  1. **That contain the same information as R**,
  2. **but are free of redundancy**.

Examples

1. We decompose the relation:

   `Recipe(ingredient, category, quantity, course, type)`

   into:

   (a) `Recipe1(ingredient, category)`, is about ingredients;

   (b) `Recipe2(course, type)`, is about cooking recipes.

   (c) `Recipe3(ingredient, course, quantity)`, is about the relationship between ingredients and courses.

2. We decompose the relation:

   `Agrifood(country, surface, year, agriarea, olives, tomatoes, wine)`

   into:

   (a) `Agrifood1(country, surface)`, concern with geographical information;

   (b) `Agrifood2(country, year, agriarea, olives, tomatoes, wine)` is about agricultural productions.

3. More, we decompose the relation:

   `Author(surname, name, dateBirth, cityBorn, nationBorn)`

   into:

   (a) `Author1(cityBorn, nationBorn)`, concern with geographical information;

   (b) `Author2(surname, name, dateBirth, cityBorn)` is about author personal information.

Third Normal Form

There is one set of Functional Dependencies that causes trouble when we decompose. Considering the relation and the given Functional Dependencies:

`Attendance(student, course, room no.)`

(1) `student` → `room no.`

(2) `course` → `room no.`

Student	**Course**	**Room no.**
Weierstrass	Statistics	B200
Lagrange	Record Linkage	A100
Lagrange	Databases	A100
Binet	Maths	A100
Binet	Databases	A100

Whenever we decompose into relations:

Student	**Room no.**
Weierstrass	B200
Lagrange	A100
Binet	A100

Course	**Room no.**
Statistics	B200
Record Linkage	A100
Databases	A100
Maths	A100

Join (natural) on tuples with equal room no., yields:

Student	**Course**	**Room no.**
Weierstrass	Statistics	B200
Lagrange	Record Linkage	A100
Lagrange	Databases	A100
Lagrange	Maths	A100
Binet	Record Linkage	A100
Binet	Databases	A100
Binet	Maths	A100

Although no Functional Dependencies were violated in the decomposed relations, is violated by the database as a whole (= database instance).

Definition: A relation R is in Third Normal Form (3NF) whenever $A_1A_2...A_n \rightarrow B$ is a non trivial FD for R, either $A_1A_2...A_n$ is a superkey or B is a member of some key.

Exercises

1. Identify FDs and decompose the relation "vegetable".

   `Vegetable(name, colour, type, season)`

   (a) `Vegetable1(name, colour, type)`

   (b) `Vegetable2(type, season)`

2. Explore daily quotes of a given cripto-currency, namely **BTC-EUR**. For this purpose we have gathered data from a financial data bank.

```
Date,Open,High,Low,Close,Adj Close,Volume
2017-01-01,913.700012,953.440002,908.640015,946.469971,946.469971,7639127
2017-01-02,946.469971,984.359985,944.349976,973.479980,973.479980,15365157
2017-01-03,973.479980,991.489990,960.640015,990.369995,990.369995,10875964
.........
```

   Try to assign a meaning at each value and then find functional dependencies (if there exists).

   Aiming to have all information associated with, we add to each line the **code BTC-EUR**. Then among various functional dependencies we suggest:

   Code, *Date* $\rightarrow$ *Volume*, *Close*

3. The election of the president of the cultural association held last week and you registered the results of each session (consider that 5 sessions has been held!) arranging data in the following table (relations). Reflecting a little bit you observed that **anomalies can occur**. Which ones? Maybe a normal form should be a solution. Describe how you normalize the relation. [**Explanation,** type: for presidential elections, nr.: for sessions, ...].

type	date	nr.	surname	name	birth-date	votes

   The relation should be split in three relations. Artificial keys are added.

   Candidate(cID, surname, name, birth-date)

   Session(code, type, date, nr.)

   Session_Candidate(code, cID, votes)