Data Management for Data Science
Sapienza Università di Roma
2015/2016

SQL

Maurizio Lenzerini, Riccardo Rosati

Dipartimento di Ingegneria informatica, automatica e gestionale
“Antonio Ruberti”
Sapienza Università di Roma

SQL

1. Data definition

1. **Data definition**
2. Data manipulation
3. Queries
4. Further aspects

SQL

- SQL ("**S**tructured **Q**uery **L**anguage") contains both the DDL (Data Definition Language) and the DML (Data Manipulation Language)
- Different versions of the language exist
- Brief history:
  - First proposal: **SEQUEL** (IBM Research, 1974);
  - first implementations in SQL/DS (IBM) and Oracle (1981);
  - from 1983 ca., “standard de facto”
  - standard versions released in 1986, 1989, **1992**, 1999, 2003
  - Last versions only partially supported by commercial systems

Using an SQL-based DBMS

- An SQL-based database management system (DBMS) is a server that allows for managing a set of relational databases
- Following the relational model, an SQL database is characterized by a schema (intensional level) and by an instance (extensional level)
- In addition, an SQL database is characterized by a set of meta-data (catalog)

Data definition in SQL

- The most important statement of the SQL DDL is

  create table

  – Defines a relation schema (specifying attributes and constraints)

  – Creates an empty instance of the relation schema

- Syntax:

```
create table TableName (
        AttributeName Domain [ Constraints ]
        ........
        AttributeName Domain [ Constraints ]
        [ OtherConstraints]
)
```

create table: example

table name

```
create table Employee (
  ID              character(6) primary key,
  Name            character(20) not null,
  Surname         character(20) not null,
  Depart          character(15),
  Salary          numeric(9) default 0,
  City            character(15),
  foreign key(Depart) references
     Department(DepName),
  unique (Surname,Name)
)
```

constraint

attribute name

domain (type)

SQL and the relational model

- **Remark**: an SQL table is defined as a **multiset** of n-tuples
- Only if the table has a primary key (or a set of attributes defined as unique), the same n-tuple cannot appear twice in the table

Domains for attributes

- **Predefined domains**
  - **Character**:
    - `char(`*n*`)` or `character(`*n*`)`
    - `varchar(`*n*`)` (or `char varying(n)`)
    - `nchar(`*n*`)` and `nvarchar(`*n*`)` (or `nchar varying(n)`) (UNICODE)
  - **Numeric**:
    - `int` or `integer`, `smallint`
    - `numeric`, (or `numeric(`*p*`)`, `numeric(`*p*`,`*s*`)`)
    - `decimal`, (or `decimal(`*p*`)`, `decimal(`*p*`,`*s*`)`)
    - `float`, `float(`*p*`)`, `real` , `double precision`
  - **Date, time:**
    - `Date`, `time`, `timestamp`
    - `time with timezone`, `timestamp with timezone`
  - **Bit:**
    - `bit(`*n*`)`
    - `bit varying(`*n*`)`
  - **Further domanis (introduced in SQL:1999)**
    - `boolean`
    - `BLOB`, `CLOB`, `NCLOB` (binary/character large object)

Domains for attributes

- **User-defined domains**
  - Sintassi

    create domain *NewDomainName*
    as *PreExistingDomain* [*Default*] [*Constraints*]

  - *Example*:

```
create domain Grade
as smallint default null
check ( value >=18 and value <= 30 )
```

Intra-relational constraints

- `not null` (over single attributes)

- `unique`: defines a set of attributes as a super-key:
  - single attribute:

    `unique` after the domain specification
  - Multiple attributes:

    `unique` (*Attribute*, ..., *Attribute*)

- `primary key`: (only one primary key can be defined on a relation) syntax similar to `unique`; implies `not null`

- `check`, for more complex constraints

Example

```
create table Employee (
  ID            character(6) primary key,
  Name          character(20) not null,
  Surname       character(20) not null,
  Depart        character(15),
  Salary        numeric(9) default 0,
  City          character(15),
  foreign key(Depart)references
     Department(DepName),
  unique (Surname,Name)
)
```

primary key, alternative

```
create table Employee (
   ID character(6) primary key,
   ...
)
```

oppure

```
create table Employee (
   ID character(6),
   ...
   primary key (ID)
)
```

Keys over multiple attributes

```
create table Employee ( ...
  Name    character(20) not null,
  Surname character(20) not null,
  unique (surname,name)
)
```

is **different** from:

```
create table Employee ( ...
  Name    character(20) not null unique,
  Surname character(20) not null unique
)
```

Inter-relational constraints

- **check**, for complex constraints
- **references** and **foreign key** allow for defining **referential** integrity constraints

  Syntax:

  – single attribute:

  references after the specification of the domain

  – multiple attributes:

  foreign key (*Attribute*, ..., *Attribute*) **references** ...

  The attributes referenced in the end table must constitute a key (**primay key** or **unique**). If they are missing, the attributes of the primary key are considered.

  Semantics: every combination (without NULL) of values for the attributes in the starting table must appear in the end table

Inter-relational constraints: example

Infringements

Code	Date	Policeman	Prov	Number
34321	1/2/95	3987	MI	39548K
53524	4/3/95	3295	TO	E39548
64521	5/4/96	3295	PR	839548
73321	5/2/98	9345	PR	839548

Policemen

ID	Surname	Name
3987	Rossi	Luca
3295	Neri	Piero
9345	Neri	Mario
7543	Mori	Gino

Inter-relational constraints: example (cont.)

Infringements

Code	Date	Policeman	Prov	Number
34321	1/2/95	3987	MI	39548K
53524	4/3/95	3295	TO	E39548
64521	5/4/96	3295	PR	839548
73321	5/2/98	9345	PR	839548

Car

Prov	Number	Surname	Name
MI	39548K	Rossi	Mario
TO	E39548	Rossi	Mario
PR	839548	Neri	Luca

Inter-relational constraints: example

```
create table Infringements (
  Code      character(6) not null primary key,
  Date      date not null,
  Policeman integer not null
           references Policemen(ID),
  Prov     character(2),
  Number    character(6),
  foreign key(Prov, Number)
          references Car(Prov,Number)
)
```

Schema modification : alter table

alter table: allows for modifying a table

Example:

```
create table Infringements (
  Code    character(6) not null primary key,
  Date          date not null,
  Policeman     integer not null
        references Policemen(ID),
  Prov    character(2),
  Number        character(6),
)

alter table Infringements
 add constraint MyConstraint foreign key(Prov, Number)
  references Car(Prov, Number)
```

It can be used to realize **cyclic** referential integrity constraints

Schema modification: `drop table`

`drop table`: eliminates a table

Syntax:

```
drop table TableName restrict | cascade
```

Esempio:

`drop table Infringements restrict` or simply

`drop table Infringements`

– eliminates the table if it is not referenced

`drop table Infringements cascade` – eliminates the table and all the tables (and the other database objects) referring to it

Definition of indices

- Is very important for the system performance
- Deals with the physical level of the DB, not the logical one
- **`create index`**
- Syntax (simplified):

```
create [unique] index IndexName on
         TableName Attribute,...,Attribute)
```

- *Example:*

```
create index IndiceIP on
         Infringements(Prov)
```

Catalog (or data dictionary)

Every DBMS creates and maintains special tables that collect the meta-data about

- **tables**
- **attributes**
- **...**

For instance, the `Columns` table contains the attributes

- `Column_Name`
- `Table_name`
- `Ordinal_Position`
- `Column_Default`
- …

SQL

2. Data manipulation

1. Data definition
2. **Data manipulation**
3. Queries
4. Further aspects

Update operations in SQL

- Update operations:
  - addition: `insert`
  - elimination: `delete`
  - modification: `update`
- Of one or multiple tuples of a relation
- Based on a condition that may involve the relation and/or other relations

Insert: syntax

```
insert into Table [ ( Attributes ) ]
       values( Values )
```

or

```
insert into Table [ ( Attributes ) ]
       select ...
```

Insert: example

```
insert into person values('Mario',25,52)


insert into person(name, age, income)
   values('Pino',25,52)


insert into person(name, income)
   values('Lino',55)



insert into person (name)
   select father
   from isFather
   where  father not in (select name from person)
```

Insert: comments

- The order ot the attributes and the values (if present) is significant
- The list of attributes and the list of values must have the same number of elements
- If the attribute list is missing, all the attributes of the relation are considered, according to the order in which they have been defined
- If the attribute list does not contain all the attributes of the relation, a null value is inserted for every missing attribute (or a default value, if declared)

Tuple elimination

Syntax:

delete from *Table* [**where** *Condition*]

Example:

```
delete from person
where age < 35

delete from isFather
where child not in
             (select name from person)
```

Delete: comments

- Deletes the tuples satisfying the condition
- It may cause (if the referential integrity constraints are defined using **cascade**) deletions in other relations
- remember: if the `where` clause is omitted, it is considered as `where true`

Tuple modification

- **Syntax**:

`update` *TableName*
`set` *Attribute* **`=`** < *Expression* | **`select`** … | **`null`** | **`default`** >
[**`where`** *Condition*]

- **Semantics**: the tuples that satisfy the «where» condition are deleted

- *Examples*:

```
update person set income = 45
where  name = 'Piero'

update person set income = income * 1.1
where  age < 30
```

SQL

3. Queries

1. Data definition
2. Data manipulation
3. **Queries**
4. Further aspects

The `select` statement (basic version)

- The query statement in SQL is

  `select`

- It defines a query and returns the result as a table

```
select   Attribute … Attribute
from     Table … Table
[where   Condition]
```

- The three sections of the statement are usually called:
  - **target list**
  - **from clause**
  - **where clause**

isMother

mother	child
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

isFather

father	child
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

person

name	age	income
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	30
Luigi	50	40
Franco	60	20
Olga	30	41
Sergio	85	35
Luisa	75	87

Selection and projection

Name and income of pepole who are less than 30 years old:

$$\mathrm{PROJ}_{\mathrm{name,\ income}}(\mathrm{SEL}_{\mathrm{age<30}}(\mathrm{person}))$$

```
select person.name, person.income
from   person
where  person.age < 30
```

name	income
Andrea	21
Aldo	15
Filippo	30

Name conventions

- To avoid ambiguity, every attribute name is composed of

TableName.AttributeName

- When there is no ambiguity, *TableName* can be omitted

```
select person.name, person.income
from   person
where  person.age < 30
```

can be written as follows:

```
select name, income
from   person
where  age < 30
```

SELECT, abbreviations

```
select person.name, person.income
from   person
where  person.age < 30
```

can be also written as:

```
select p.name as name, p.income as income
from   person as p
where  p.age < 30
```

or:

```
select p.name as name, p.income as income
from   person p
where  p.age < 30
```

Projection

surname and city of all employees

employees

ID	surname	city	salary
7309	Neri	Napoli	55
5998	Neri	Milano	64
9553	Rossi	Roma	44
5698	Rossi	Roma	64

$\text{PROJ}_{\text{surname, city}}$ (employees)

Projection and duplicates

```
select surname,
       city
from employees
```

```
select distinct surname,
                city
from employees
```

surname	city
Neri	Napoli
Neri	Milano
Rossi	Roma
Rossi	Roma

surname	city
Neri	Napoli
Neri	Milano
Rossi	Roma

SELECT, usage of "as"

"as" is used in the attribute list to specify a name for an attribute of the result. If such a name is not specified, then the attribute name of the result is equal to the corresponding attribute of the input table.

Example:

```
select name as personName, income as salary
from   person
where  age < 30
```

returns a relation with two attributes: `personName` and `salary`

```
select name, income
from   person
where  age < 30
```

returns a relation with two attributes: `name` and `income`

Exercise 1

Compute the table obtained from table **`person`** selecting only the people whose income is between 20 and 30, and adding an attribute that has the same value as the attribute **`income`** in every tuple

Show the result of the query over the table **`person`** shown at page 32.

person

name	age	income

Solution, Exercise 1

```
select name, age, income,
       income as repeatedIncome
from   person
where  income >= 20 and income <= 30
```

name	age	income	repeatedIncome
Andrea	27	21	21
Filippo	26	30	30
Franco	60	20	20

Selection, without projection

name, age and income of people who are less than 30 years old

$$\mathrm{SEL}_{age<30}(\mathrm{person})$$

```
select  *
from    person
where   age < 30
```

Is an abbreviation for:

all attributes

```
select  name,age,income
from    person
where   age < 30
```

Projection, without selection

name and income of all people:

$$\mathrm{PROJ}_{\mathrm{name,\ income}}(\mathrm{person})$$

```
select  name,  income
from    person
```

Is an abbreviation for:

```
select  p.name,  p.income
from    person p
where   true
```

Expressions in the target list

```
select income/2 as semesterIncome
from   person
where  name = 'Luigi'
```

Complex condition in the “where” clause:

```
select *
from   person
where  income > 25
       and (age < 30 or age > 60)
```

“LIKE” condition

People having a name whose first letter is '**A**’, and whose third letter is '**d**‘:

```
select *
from   person
where  name like 'A_d%'
```

Null values

Employees whose age is or might be greater than 40:

$$\text{SEL}_{\text{age} > 40 \text{ OR age IS NULL}} \text{ (employees)}$$

```
select *
from   employees
where   age > 40 or age is null
```

Exercise 2

Compute the tble obtained from table `employees` selecting only the ones whose city is Roma or Milano, projecting the data on the attribute `salary`, and adding an attribute having, in every tuple, a value that is the double the value of the attribute `salary`

Show the result of the query over the table shown at page 36.

employees

ID	surname	city	salary

Solution, Exercise 2

```
select salary,
       salary*2 as doubleSalary
from   employees
where  city = 'Milano' or
       city = 'Roma'
```

salary	doubleSalary
64	128
44	88
64	128

Selection, projection and join

- the `select` statements with a single relation in the `from` clause allow for expressing:
  - selections
  - projections
  - renamings

- **joins** (and cartesian products) are expressed using two or more relations in the `from` clause

SQL and relational algebra

Given the relations R1(A1,A2) and R2(A3,A4):

The semantics of the query

```
select R1.A1, R2.A4
from   R1, R2
where  R1.A2 = R2.A3
```

can be described as a combination of:

- cartesian product (`from`)
- selection (`where`)
- projection (`select`)

Remark: this does not mean that the DBMS necessarily computes the cartesian product to answer the query!

SQL: DBMS execution of queries

- SQL expressions are declarative, and we are describing their semantics
- In pratice, DBMSs execute operations in efficient ways, for instance:
  - They execute selections as soon as possible
  - If possible, they execute joins instead of cartesian product
- The ability of DBMSs to optimize queries makes it usually not necessary to deal with efficiency when a query is specified

SQL and relational algebra, 2

Given the relations R1(A1,A2) and R2(A3,A4)

```
select R1.A1, R2.A4
from   R1, R2
where  R1.A2 = R2.A3
```

corresponds to :

$$\text{PROJ}_{A1,A4}\ (\text{SEL}_{A2=A3}\ (\text{R1 JOIN R2}))$$

SQL and relational algebra, 3

Renamings may be necessary:

- in the target list
- In the from clause (cartesian product), in particular when the same table must be referred multiple times

```
select X.A1 as B1, ...
from   R1 X, R2 Y, R1 Z
where  X.A2 = Y.A3 and ...
```

can be written as

```
select X.A1 as B1, ...
from   R1 as X, R2 as Y, R1 as Z
where  X.A2 = Y.A3 and ...
```

SQL and relational algebra: example

```
select X.A1 as B1, Y.A4 as B2
from   R1 X, R2 Y, R1 Z
where  X.A2 = Y.A3 and Y.A4 = Z.A1
```

$$\mathrm{REN}_{B1,B2 \leftarrow A1,A4}\,($$

$$\mathrm{PROJ}_{A1,A4}\,(\mathrm{SEL}_{A2\,=\,A3\ \mathbf{and}\ A4\,=\,C1}($$

$$\mathrm{R1\ JOIN\ R2\ JOIN\ REN}_{C1,C2\,\leftarrow\,A1,A2}\,(\mathrm{R1}))))$$

isMother

mother	child
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

Is Father

father	child
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

person

name	age	income
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	30
Luigi	50	40
Franco	60	20
Olga	30	41
Sergio	85	35
Luisa	75	87

Exercise 3: selection, projection and join

Return the fathers of people who earn more than 20 millions.

Exercise 3: solution

Return the fathers of people who earn more than 20 millions.

$\text{PROJ}_{\text{father}}(\text{isFather JOIN}_{\text{child=name}}\ \text{SEL}_{\text{income>20}}\ (\text{person}))$

```
select  distinct isFather.father
from    person, isFather
where   isFather.child = person.name
        and person.income > 20
```

Exercise 4: join

Return the father and the mother of every person.

Exercise 4: solution

Return the father and the mother of every person.

This can be expressed in relational algebra through the **natural join**.

isFather JOIN isMother

In SQL:

```
select  isMother.child, father, mother
from    isMother, isFather
where   isFather.child = isMother.child
```

Exercise 4: solution

If we interpret the question as: return father and mother of every person appearing in the «person» table, then we need an additional join:

In relational algebra:

$$\text{PROJ}_{\text{child,father,mother}}\ ((\text{isMother JOIN isFather})\ \text{JOIN}_{\text{child=name}}\ \text{person})$$

In SQL:

```
select  isMother.child, father, mother
from    isMother, isFather, person
where   isFather.child = isMother.child
        and isMother.child = person.name
```

Exercise 5: join and other operations

Return the persons earning more than their fathers, showing name, income and father’s income.

Exercise 5: solution

Return the persons earning more than their fathers, showing name, income and father's income.

$$\text{PROJ}_{\text{name, income, RP}}\,(\text{SEL}_{\text{income>RP}}\,(\text{REN}_{\text{NP,EP,RP} \leftarrow \text{name,age,income}}(\text{person})\ \text{JOIN}_{\text{NP=father}}\ (\text{isFather JOIN}_{\text{child =name}}\ \text{person})))$$

```
select    c.name, c.income, p.income
from      person p, isFather t, person c
where     p.name = t.father and
          t.child = c.name and
          c.income > p.income
```

SELECT, with renaming of the result

Return the persons earning more than their fathers, showing name, income and father’s income.

```
select child, c.income as income,
       p.income as fatherIncome
from   person p, isFather t, person c
where  p.name = t.father and
       t.child = c.name and
       c.income > p.income
```

SELECT with explicit join

```
select …
from Table { join Table on JoinCondition }, …
[ where OtherCondition ]
```

this is the SQL operator corrisponding to theta-join

Explicit join

Return the father and the mother of every person:

```
select isFather.child, father, mother
from   isMother, isFather
where  isFather.child = isMother.child
```

```
select mother, isFather.child, father
from   isMother join isFather on
       isFather.child = isMother.child
```

explicit join

Exercise 6: explicit join

Return the persons earning more than their fathers, showing name, income and father’s income.

Express the query using the explicit join.

SELECT with explicit join: example

Return the persons earning more than their fathers, showing name, income and father’s income.

```
select c.name, c.income, p.income
from   person p, isFather t, person c
where  p.name = t.father and
       t.child = c.name and
       c.income > p.income
```

Using the explicit join:

```
select c.name, c.income, p.income
from person p join isFather t on p.name=t.father
     join person c on t.child=c.name
where  c.income > p.income
```

Natural join (less frequently used)

$$\text{PROJ}_{\text{child,father,mother}}(\text{isFather JOIN }_{\text{child}\leftarrow\text{name}}\ \text{REN }_{\text{name}\leftarrow\text{child}}(\text{isMother}))$$

In algebra: isFather JOIN isMother

In SQL (with Explicit join):

```
select isFather.child, father, mother
from  isMother join isFather on
      isFather.child = isMother.child
```

In SQL (with natural join) :

```
select isFather.child, father, mother
from isMother natural join isFather
```

Left outer join

Return every pair child/father and, if known, the mother.

```
select isFather.child, father, mother
from   isFather left outer join isMother
       on isFather.child = isMother.child
```

(if the mother does not exist, a null value is returned)

Remark: “`outer`” is optional

```
select isFather.child, father, mother
from   isFather left join isMother
       on isFather.child = isMother.child
```

Right outer join

if we use the **right** outer join:

```
select isFather.child, father, mother
from   isFather right outer join isMother
       on isFather.child = isMother.child
```

the query returns **all** mothers (even those who do not have a join with isFather)

Left and right outer join: examples

```
select isFather.child, father, mother
from   isMother join isFather
       on isMother.child = isFather.child

select isFather.child, father, mother
from   isMother left outer join isFather
       on isMother.child = isFather.child

select isFather.child, father, mother
from   isMother right outer join isFather
       on isMother.child = isFather.child
```

Full outer join: examples

```
select isFather.child, father, mother
from   isMother full outer join isFather
       on isMother.child = isFather.child
```

```
select name, father, mother
from   person full outer join isMother on
       person.name = isMother.child
       full outer join isFather on
       person.name = isFather.child
```

Ordering the result: `order by`

name and income of persons whose age is less than 30 in alphabetical order

```
select name, income
from   person
where  age < 30
order by name
```

ascending order

```
select name, income
from   person
where  age < 30
order by name desc
```

descending order

Ordering the result: `order by`

```
select name, income
from   person
where  age < 30
```

name	income
Andrea	21
Aldo	15
Filippo	30

```
select name, income
from   person
where  age < 30
order by name
```

name	income
Aldo	15
Andrea	21
Filippo	30

Bounding the size of the query result

```
select name, income
from   person
where  age < 30
order by name
limit 2
```

Bounding the size of the query result

```
select name, income
from   person
where  age < 30
order by name
limit 2
```

name	income
Andrea	21
Aldo	15

Aggregate operators

The target list may contain expressions that compute values based on sets of tuples:

- count, min, max, average, total

(simplified) syntax:

Function **(** [**`distinct`**] *ExpressionOverAttributes* **)**

Aggregate operators: `count`

Syntax:

- Count the number of tuples:

  `count (*)`

- Count the values in an attribute:

  `count (`*Attributo*`)`

- Count the **distinct** values in an attribute:

  `count (distinct` *Attributo*`)`

`count`: example and semantics

Esempio: Return the number of children of Franco:

```
select  count(*)  as  NumChildrenFranco
from    isFather
where   father = 'Franco'
```

Semantics: the aggregate operator (`count`) is applied to the result of the following query:

```
select  *
from    isFather
where   father = 'Franco'
```

count: example

isFather

father	child
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

NumChildrenFranco
2

`count` and null values

```
select count(*)
from   person
```

Result = 4

```
select count(income)
from   person
```

Result = 3

```
select count(distinct income)
from   person
```

Result = 2

person

name	age	income
Andrea	27	21
Aldo	25	NULL
Maria	55	21
Anna	50	35

Other aggregate operators

sum, **avg**, **max**, **min**

- Allow an attribute or an expression as argument (not “*****”)
- **sum** and **avg**: numeric or date/time arguments
- **max** and **min**: arguments on which a total ordering is defined

Esempio: return the income average of Franco’s children:

```
select avg(income)
from   person join isFather on
       name = child
where  father = 'Franco'
```

Aggregate operators and null values

```
select avg(income) as averageIncome
from   person
```

person

name	age	income
Andrea	27	30
Aldo	25	NULL
Maria	55	36
Anna	50	36

this tuple is ignored

averageIncome
34

Aggregate operators and target list

The following query does not make sense:

```
select name, max(income)
from   person
```

For the query to make sense, the **target list** must be **homogeneous**, for instance:

```
select min(age), avg(income)
from   person
```

Aggregate operators and grouping

- In the previous cases, the aggregate operators were applied to all the tuples constituting the query result

- In many cases, we want the aggregate functions to be applied to **partitions of tuples**

- To specify such partitions, the clause `group by` can be used:

```
group by AttributeList
```

Aggregate operators and grouping

Return the number of children of every father:

```
select  father, count(*) as NumChildren
from    isFather
group by father
```

isFather

father	child
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

father	NumChildren
Sergio	1
Luigi	2
Franco	2

Semantics of queries with aggregate operators and grouping

1. execute the query **ignoring the `group by` clause** and the aggregate operators:

   ```
   select *
   from   isFather
   ```

2. group the **tuples having the same values for the attributes mentioned in the `group by` clause**

3. the aggregate operator is applied to every group and a tuple is produced for every group

Exercise 7: group by

Return the maximum income (and the age) of every group of persons whose age is greater than 18 and have the same age.

person

name	age	income

Exercise 7: solution

Return the maximum income (and the age) of every group of persons whose age is greater than 18 and have the same age.

```
select age, max(income)
from   person
where  age > 18
group by age
```

Grouping and target list

In query that uses the **`group by`** clause, the taget list should be «homogeneous», namely, **only** attributes appearing in the **`group by`** clause and aggregation functions should appear in the list.

Example:

- Income of persons, grouped by age (**non-homogeneous** target list):

```
select age, income
from person
group by age
```

- Average income of persons, grouped by age (**homogeneous**, target list (in every group there is only one average income):

```
select age, avg(income)
from person
group by age
```

Non-homogeneous target list

What happens if the target list is non-homogeneous?

Some systems do not raise any error and for each group return one of the values associated with the group.

Example:

Income of persons, grouped by age:

```
select age, income
from person
group by age
```

The DBMS MySQL, for instance, does not raise any error: for each group, it chooses one of the incomes appearing in the group and returns such a value as the income attribute of the target list.

Conditions on groups

We can also impose **selection conditions on groups**. Group selection is **obviously different** from the condition that selects the tuples forming the groups (`where` clause).

Group selection is realized by the **`having`** clause, which must appear after the "`group by`" clause.

Example: return the fathers whose children have an average income greater than 25.

```
select  father, avg(c.income)
from    person c join isFather
        on child = name
group by father
having avg(c.income) > 25
```

Exercise 8: `where` or `having`?

Return the fathers whose children under 30 have an average income greater than 20.

Exercise 8: solution

Return the fathers whose children under 30 have an average income greater than 20.

```
select father, avg(c.income)
from   person c join isFather
       on child = name
where  c.age < 30
group by father
having avg(c.income) > 20
```

Syntax of select statement (summary)

SelectSQL ::=

```
select     AttributeOrExpressionList
from       TableList
[ where    SimpleConditions ]
[ group by GroupingAttributeList ]
[ having   AggregationConditions ]
[ order by OrderingAttributeList ]
[ limit    Number ]
```

Union, intersection and difference

A single **select** statement does not allow for expressing unions (for instance, the union of two tables)

An explicit statement is needed:

```
select ...
union [all]
select ...
```

With **union**, duplicate tuples are eliminated

With **union all**, duplicate tuples are kept

Positional notation

```
select father, child
from   isFather
union
select mother, child
from   isMother
```

Which are the attributes of the result? It depends on the system:

- new names established by the system
- the names of the first select statement
- ...

Union: result

father	child
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

Positional notation: example

```
select father, child
from   isFather
union
select mother, child
from   isMother
```

```
select father, child
from   isFather
union
select child, mother
from   isMother
```

These queries are different!

Positional notation

With renaming (same as before):

```
select father as parent, child
from   isFather
union
select child, mother as parent
from   isMother
```

If we want to return fathers and mothers as parents, this is the correct query:

```
select father as parent, child
from   isFather
union
select mother as parent, child
from   isMother
```

Difference

```
select name
from   employee
except
select surname as name
from   employee
```

Note: `except` eliminates duplicate tuples

Note: `except all` does not eliminate duplicate tuples

The difference can also be expressed by nested `select` statements.

Intersection

```
select name
from   employee
intersect
select surname as name
from   employee
```

is equivalent to

```
select distinct i.name
from   employee i, employee j
where  i.name = j.surname
```

Note: `intersect` eliminates duplicate tuples

Note: `intersect all` does not eliminate duplicate tuples

Nested queries

- A nested `select` statement can appear as a condition in the where clause

- In particular, the conditions allow for:
  - comparing an attribute (or a sequence of attributes) with the result of a sub-query
  - existential quantification

Nested queries: example

name and income of Franco’s father:

```
select   name, income
from     person, isFather
where    name = father and child = 'Franco'
```

```
select   name, income
from     person
where    name = (select father
                 from   isFather
                 where  child = 'Franco')
```

Nested queries: operators

The result of a nested query can be compared in the `where` clause using several **operators**:

- Equality and the other comparison operator: in this case, the result of the nested query must be a single tuple
- If the result of the nested query may contain multiple tuples, the nested query can be preceded by:
  - `any`: returns true if the comparison is true for **at least** one of the tuples in the result of the nested query
  - `all`: returns true if the comparison is true for **every** tuple in the result of the nested query
- The operator `in`, which is equivalent to `=any`
- The operator `not in`, which is equivalent to `<>all`
- The operator `exists`

Nested queries: example

name and income of the fathers of persons earning more than 20 millions:

```
select distinct p.name, p.income
from   person p, isFather, person c
where  p.name = father and child = c.name
       and c.income > 20
```

```
select name, income
from   person
where  name = any (select father
                   from   isFather, person
                   where  child = name
                          and income > 20)
```

Fathers of persons earning more than 20 millions

Nested queries: example

name and income of the fathers of persons earning more than 20 millions:

```
select name, income
from   person
where  name in (select father
                from   isFather, person
                where child = name
                and income > 20)
```

```
select name, income
from   person
where  name in (select father
                from  isFather
                where child in (select name
                                  from  person
                                  where income > 20)
```

Persons earning more than 20 millions

Fathers of persons earning more than 20 millions

Nested queries: `all` (example)

Persons whose income is greater than the income of every person who is less than 30 years old:

```
select name
from   person
where  income > all ( select income
                       from person
                       where age < 30 )
```

Nested queries: `exists` (example)

The `exists` operator is used to return true if the results of the sub-query is **not empty**.

Example: persons having at least a child.

```
select *
from   person p
where  exists (select *
               from  isFather
               where father = p.name)
       or
       exists (select *
               from  isMother
               where mother = p.name)
```

Notice that the attribute **`name`** refers to the relation in the `from` clause.

Exercise 9: nested queries

Return name and age of mothers having at least a child who is less than 18 years old.

Exercise 9: nested queries

Return name and age of mothers having at least a child who is less than 18 years old.

Solution 1: a join to select name and age of mothers, and a sub-query for the condition on the children

Solution 2: two sub-queries and no join

Exercise 9: solution 1

Return name and age of mothers having at least a child who is less than 18 years old.

```
select name, age
from   person, isMother
where  name = mother and
       child in (select name
                  from   person
                  where  age < 18)
```

Exercise 9: solution 2

Return name and age of mothers having at least a child who is less than 18 years old.

```
select name, age
from person
where name in (select mother
                from isMother
                where child in (select name
                                 from person
                                 where age<18))
```

Nested queries: comments

- Nested queries may pose performance problems to the DBMSs (since they are not very good in optimizing the execution of such statements)
- However, nested queries are sometimes more readable than equivalent, non-nested ones.
- In some systems, sub-queries cannot contain set operators, but this is not a significant limitation.

Nested queries, comments

- **visibility** rules:
  - It is not possible to refer to variables (attributes) defined in inner blocks
  - If a variable or table name is omitted, the assumption is that it refers to the «closest» variable or table

- A block can refer to varables defined in the same block or in outer blocks, unless they are hidden by definitions of variables with the same name.

- **Semantics**: the inner query is executed once **for each tuple** of the outer query

Nested queries: visibility

Return the persons having at least a child.

```
select *
from   person
where  exists (select *
               from  isFather
               where father = name)
       or
       exists (select *
               from  isMother
               where mother = name)
```

Attribute `name` refers to the relation `person` in the `from` clause.

Nested queries: visibility

The following query is incorrect:

```
select *
from employee
where depart in (select name
                 from department D1
                 where name = 'Produzione')
      or
      depart in (select name
                 from department D2
                 where D2.citta = D1.citta)
```

employee	name	surname	depart
department	name	address	city

Example

name and income of the fathers of persons earning more than 20 millions, **returning the child’s income too**.

```
select distinct p.name, p.income, c.income
from   person p, isFather, person c
where  p.name = father and child = c.name
       and c.income > 20
```

In this case the following “intuitive” nested query is not correct:

```
select name, income, c.income
from person
where name in (select father
               from isFather
               where child in (select name
                                from person c
                                where c.income > 20))
```

Nested and correlated queries

It may be necessary to use, in an inner block, variables defined in outer blocks: in this case the query is called nested and **correlated**.

Example: fathers whose children earn more than 20 millions.

```
select distinct father
from isFather z
where not exists (select *
                  from isFather w, person
                  where w.father = z.father
                        and w.child = name
                        and income <= 20)
```

Exercise 10: nested and correlated queries

Return name and age of every mother having at least a child who is less than 30 years younger than her.

Exercise 10: solution

Return name and age of every mother having at least a child who is less than 30 years younger than her.

```
select name, age
from   person p, isMother
where  name = mother and
       child in (select name
                 from   person
                 where  p.age - age < 30)
```

Difference can be expressed by nested queries

```
select name from employee
  except
select surname as name from employee
```

```
select name
from employee
where name not in (select surname
                   from employee)
```

Intersection can be expressed by nested queries

```
select name from employee
  intersection
select surname from employee
```

```
select name
from employee
where  name in (select surname
                from employee)
```

Exercise 11: nesting and functions

Return the person(s) with maximum income.

Exercise 11: solution

Return the person(s) with maximum income.

```
select *
from person
where income = (select max(income)
                from person)
```

or:

```
select *
from person
where income >= all (select income
                     from person)
```

Nested queries: condition on multiple attributes

Return the persons whose pair (age, income) is different from all other persons.

```
select *
from person p
where (age,income) not in
                          (select age, income
                           from person
                           where name <> p.name)
```

Nested queries in the `from` clause

Nested queries may appear not only in the `where` clause, but also in the `from` clause:

```
select p.father
from isFather p, (select name
                  from person
                  where age > 30) c
where c.name = p.child
```

Semantics: the table whose alias is `f`, and defined as a nested query in the `from` clause, is not a database table, but is computed using the associated `select` query.

SQL

4. Further aspects

1. Data definition
2. Data manipulation
3. Queries
4. **Further aspects**

Generic integrity constraints: check

To specify complex constraints on a tuple or a table:

`check` (*Condition*)

```
create table employee
( ID character(6),
  surname character(20),
  name character(20),
  sex character not null check (sex in (‘M’,‘F’))
  salary integer,
  manager character(6),
  check (salary <= (select salary
                       from   employee j
                       where  manager = j.ID))
)
```

Views

- A view is a table **whose instance is derived from other tables through a query**.

create view *ViewName* [**(***AttributeList***)**] **as** *SelectSQL*

- Views are virtual tables: their instance is computed only when they are used by other queries.

- *Example*:

```
create view adminEmp(Mat,name,surname,Stip) as
   select ID, name, surname, salary
   from   employee
   where  Depart = 'Administration' and
          salary > 10
```

Nested queries in the `having` clause

- Return the age of persons such that sum of the income of persons having that age is maximum.
- Assuming there are no null values in the income attribute, and using a nested query in the **`having`** clause:

```
select age
from   person
group by age
having sum(income) >= all (select sum(income)
                            from person
                            group by age)
```

Solution with views

```
create view ageincome(age,total-income) as
   select age, sum(income)
   from   person
   group by age


select age
from   ageincome
where  total-income = (select max(total-income)
                         from ageincome)
```