Data Management for Data Science
Sapienza Università di Roma
2019/2020

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 ("Structured Query Language") 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”
  – 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] )`
```sql
CREATE TABLE 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),  
    Department (DepName),  
    unique (Surname, Name)
)
```
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 domains (introduced in SQL:1999)**
    • boolean
    • BLOB, CLOB, NCLOB (binary/character large object)
Domains for attributes

- **User-defined domains**
  - Sintassi
    
    ```sql
    create domain NewDomainName
    as PreExistingDomain [ Default ] [ Constraints ]
    ```

  - **Example**:
    ```sql
    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
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 (**primary 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Date</th>
<th>Policeman</th>
<th>Prov</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>34321</td>
<td>1/2/95</td>
<td>3987</td>
<td>MI</td>
<td>39548K</td>
</tr>
<tr>
<td>53524</td>
<td>4/3/95</td>
<td>3295</td>
<td>TO</td>
<td>E39548</td>
</tr>
<tr>
<td>64521</td>
<td>5/4/96</td>
<td>3295</td>
<td>PR</td>
<td>839548</td>
</tr>
<tr>
<td>73321</td>
<td>5/2/98</td>
<td>9345</td>
<td>PR</td>
<td>839548</td>
</tr>
</tbody>
</table>

Policemen

<table>
<thead>
<tr>
<th>ID</th>
<th>Surname</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3987</td>
<td>Rossi</td>
<td>Luca</td>
</tr>
<tr>
<td>3295</td>
<td>Neri</td>
<td>Piero</td>
</tr>
<tr>
<td>9345</td>
<td>Neri</td>
<td>Mario</td>
</tr>
<tr>
<td>7543</td>
<td>Mori</td>
<td>Gino</td>
</tr>
</tbody>
</table>
### Inter-relational constraints: example (cont.)

#### Infringements

<table>
<thead>
<tr>
<th>Code</th>
<th>Date</th>
<th>Policeman</th>
<th>Prov</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>34321</td>
<td>1/2/95</td>
<td>3987</td>
<td>MI</td>
<td>39548K</td>
</tr>
<tr>
<td>53524</td>
<td>4/3/95</td>
<td>3295</td>
<td>TO</td>
<td>E39548</td>
</tr>
<tr>
<td>64521</td>
<td>5/4/96</td>
<td>3295</td>
<td>PR</td>
<td>839548</td>
</tr>
<tr>
<td>73321</td>
<td>5/2/98</td>
<td>9345</td>
<td>PR</td>
<td>839548</td>
</tr>
</tbody>
</table>

#### Car

<table>
<thead>
<tr>
<th>Prov</th>
<th>Number</th>
<th>Surname</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MI</td>
<td>39548K</td>
<td>Rossi</td>
<td>Mario</td>
</tr>
<tr>
<td>TO</td>
<td>E39548</td>
<td>Rossi</td>
<td>Mario</td>
</tr>
<tr>
<td>PR</td>
<td>839548</td>
<td>Neri</td>
<td>Luca</td>
</tr>
</tbody>
</table>
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:*

```sql
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),
)
```

```sql
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):
  ```sql
  create [unique] index IndexName on TableName Attribute,...,Attribute
  ```
- **Example:**
  ```sql
  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

\[
\text{insert into } Table[(\text{Attributes})] \\
\text{values(Values)}
\]

or

\[
\text{insert into } Table[(\text{Attributes})] \\
\text{select } ...
\]
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 of 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:**

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

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

• **Examples:**

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

```sql
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
### person

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>27</td>
<td>21</td>
</tr>
<tr>
<td>Aldo</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>Maria</td>
<td>55</td>
<td>42</td>
</tr>
<tr>
<td>Anna</td>
<td>50</td>
<td>35</td>
</tr>
<tr>
<td>Filippo</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>Luigi</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>Sergio</td>
<td>60</td>
<td>20</td>
</tr>
<tr>
<td>Olga</td>
<td>30</td>
<td>41</td>
</tr>
<tr>
<td>Sergio</td>
<td>85</td>
<td>35</td>
</tr>
<tr>
<td>Luisa</td>
<td>75</td>
<td>87</td>
</tr>
</tbody>
</table>

### isMother

<table>
<thead>
<tr>
<th>mother</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luisa</td>
<td>Maria</td>
</tr>
<tr>
<td>Luisa</td>
<td>Luigi</td>
</tr>
<tr>
<td>Anna</td>
<td>Olga</td>
</tr>
<tr>
<td>Anna</td>
<td>Filippo</td>
</tr>
<tr>
<td>Maria</td>
<td>Andrea</td>
</tr>
<tr>
<td>Maria</td>
<td>Aldo</td>
</tr>
</tbody>
</table>

### isFather

<table>
<thead>
<tr>
<th>father</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sergio</td>
<td>Franco</td>
</tr>
<tr>
<td>Luigi</td>
<td>Olga</td>
</tr>
<tr>
<td>Luigi</td>
<td>Filippo</td>
</tr>
<tr>
<td>Franco</td>
<td>Andrea</td>
</tr>
<tr>
<td>Franco</td>
<td>Aldo</td>
</tr>
</tbody>
</table>
Selection and projection

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

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

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

<table>
<thead>
<tr>
<th>name</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>21</td>
</tr>
<tr>
<td>Aldo</td>
<td>15</td>
</tr>
<tr>
<td>Filippo</td>
<td>30</td>
</tr>
</tbody>
</table>
Name conventions

- To avoid ambiguity, every attribute name is composed of
  \[TableName\cdot AttributeName\]

- When there is no ambiguity, \(TableName\) can be omitted

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

can be written as follows:

```sql
select name, income
from person
where age < 30
```
**SELECT, abbreviations**

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

can be also written as:

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

or:

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

surname and city of all employees

employees

<table>
<thead>
<tr>
<th>ID</th>
<th>surname</th>
<th>city</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>7309</td>
<td>Neri</td>
<td>Napoli</td>
<td>55</td>
</tr>
<tr>
<td>5998</td>
<td>Neri</td>
<td>Milano</td>
<td>64</td>
</tr>
<tr>
<td>9553</td>
<td>Rossi</td>
<td>Roma</td>
<td>44</td>
</tr>
<tr>
<td>5698</td>
<td>Rossi</td>
<td>Roma</td>
<td>64</td>
</tr>
</tbody>
</table>

PROJ_{surname, city} (employees)
Projection and duplicates

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

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

```sql
select name as personName, income as salary
from person
where age < 30
returns a relation with two attributes: personName and salary
```

```sql
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.

<table>
<thead>
<tr>
<th>person</th>
<th>name</th>
<th>age</th>
<th>income</th>
</tr>
</thead>
</table>
Solution, Exercise 1

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

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>income</th>
<th>repeatedIncome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>27</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>Filippo</td>
<td>26</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>Franco</td>
<td>60</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>
Selection, without projection

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

\[ \text{SEL}_{\text{age}<30}(\text{person}) \]

select *  
from person  
where age < 30

Is an abbreviation for:

select name, age, income  
from person  
where age < 30
Projection, without selection

name and income of all people:

\[
\text{PROJ}_{\text{name, income}}(\text{person})
\]

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

Is an abbreviation for:

```sql
select p.name, p.income
from   person p
where  true
```
Expressions in the target list

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

Complex condition in the “where” clause:

```sql
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:

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

```sql
select *
from employees
where age > 40 or age is null
```
Exercise 2

Compute the table 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

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

<table>
<thead>
<tr>
<th>salary</th>
<th>doubleSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>44</td>
<td>88</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
</tr>
</tbody>
</table>
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
Given the relations \( R_1(A_1,A_2) \) and \( R_2(A_3,A_4) \):

The semantics of the query

\[
\text{select } R_1.A_1, R_2.A_4 \\
\text{from } R_1, R_2 \\
\text{where } R_1.A_2 = R_2.A_3
\]

can be described as a combination of:

- cartesian product (\textit{from})
- selection (\textit{where})
- projection (\textit{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 practice, 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
Given the relations \( R_1(A_1,A_2) \) and \( R_2(A_3,A_4) \)

\[
\begin{align*}
\text{select} & \quad R_1.A_1, \; R_2.A_4 \\
\text{from} & \quad R_1, \; R_2 \\
\text{where} & \quad R_1.A_2 = R_2.A_3
\end{align*}
\]

corresponds to:

\[
\text{PROJ}_{A_1,A_4} \left( \text{SEL}_{A_2=A_3} \left( R_1 \; \text{JOIN} \; R_2 \right) \right)
\]
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

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

can be written as

```sql
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

```sql
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
```

```sql
RENAME B1,B2 AS A1,A4 ( 
    PROJECTION A1,A4 ( 
        SELECTION A2 = A3 AND A4 = C1 ( 
            R1 JOIN R2 JOIN RENAME C1,C2 AS A1,A2 ( R1 )))
    )
)
### person

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>27</td>
<td>21</td>
</tr>
<tr>
<td>Aldo</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>Maria</td>
<td>55</td>
<td>42</td>
</tr>
<tr>
<td>Anna</td>
<td>50</td>
<td>35</td>
</tr>
<tr>
<td>Filippo</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td>Luigi</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>Franco</td>
<td>60</td>
<td>20</td>
</tr>
<tr>
<td>Olga</td>
<td>30</td>
<td>41</td>
</tr>
<tr>
<td>Sergio</td>
<td>85</td>
<td>35</td>
</tr>
<tr>
<td>Luisa</td>
<td>75</td>
<td>87</td>
</tr>
</tbody>
</table>

### isMother

<table>
<thead>
<tr>
<th>mother</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luisa</td>
<td>Maria</td>
</tr>
<tr>
<td>Luisa</td>
<td>Luigi</td>
</tr>
<tr>
<td>Anna</td>
<td>Olga</td>
</tr>
<tr>
<td>Anna</td>
<td>Filippo</td>
</tr>
<tr>
<td>Maria</td>
<td>Andrea</td>
</tr>
<tr>
<td>Maria</td>
<td>Aldo</td>
</tr>
</tbody>
</table>

### Is Father

<table>
<thead>
<tr>
<th>father</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sergio</td>
<td>Franco</td>
</tr>
<tr>
<td>Luigi</td>
<td>Olga</td>
</tr>
<tr>
<td>Luigi</td>
<td>Filippo</td>
</tr>
<tr>
<td>Franco</td>
<td>Andrea</td>
</tr>
<tr>
<td>Franco</td>
<td>Aldo</td>
</tr>
</tbody>
</table>
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.

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

\[
\text{select distinct isFather.father from person, isFather}
\text{where isFather.child = person.name}
\text{and person.income > 20}
\]
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.

\[
\text{isFather JOIN isMother}
\]

In SQL:

\[
\begin{align*}
\text{select } & \text{isMother.child, father, mother} \\
\text{from } & \text{isMother, isFather} \\
\text{where } & \text{isFather.child = isMother.child}
\end{align*}
\]
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:

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

In SQL:

```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.

\[
\text{select } c\text{.name, } c\text{.income, } p\text{.income from person } p, \text{ isFather } t, \text{ person } c \\
\text{where } p\text{.name }= t\text{.father and } t\text{.child }= c\text{.name and } c\text{.income }> p\text{.income}
\]
SELECT, with renaming of the result

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

```sql
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 corresponding to theta-join
Explicit join

Return the father and the mother of every person:

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

```sql
select mother, isFather.child, father
from   isMother join isFather
        on  isFather.child = isMother.child
```
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.
Return the persons earning more than their fathers, showing name, income and father’s income.

```sql
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:

```sql
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)

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

In algebra: \( \text{isFather JOIN isMother} \)

In SQL (with Explicit join):
\[
\begin{align*}
\text{select} & \quad \text{isFather.child}, \quad \text{father}, \quad \text{mother} \\
\text{from} & \quad \text{isMother join isFather} \quad \text{on} \\
& \quad \text{isFather.child} = \text{isMother.child}
\end{align*}
\]

In SQL (with natural join):
\[
\begin{align*}
\text{select} & \quad \text{isFather.child}, \quad \text{father}, \quad \text{mother} \\
\text{from} & \quad \text{isMother natural join isFather}
\end{align*}
\]
**Left outer join**

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

```sql
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

```sql
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:

```sql
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

- SQL statement 1:
  ```sql
  select isFather.child, father, mother
  from isMother full outer join isFather
  on isMother.child = isFather.child
  ```

- SQL statement 2:
  ```sql
  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: \texttt{order by} \\

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

\begin{verbatim}
select name, income 
from person 
where age < 30 
\textbf{order by name}
\end{verbatim}

\begin{verbatim}
select name, income 
from person 
where age < 30 
\textbf{order by name desc}
\end{verbatim}
Ordering the result: order by

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

Ordering the result:

```
order by name
```

<table>
<thead>
<tr>
<th>name</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>21</td>
</tr>
<tr>
<td>Aldo</td>
<td>15</td>
</tr>
<tr>
<td>Filippo</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aldo</td>
<td>15</td>
</tr>
<tr>
<td>Andrea</td>
<td>21</td>
</tr>
<tr>
<td>Filippo</td>
<td>30</td>
</tr>
</tbody>
</table>
Bounding the size of the query result

```sql
select name, income
from   person
where  age < 30
order by name
limit 2
```
Bounding the size of the query result

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

<table>
<thead>
<tr>
<th>name</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>21</td>
</tr>
<tr>
<td>Aldo</td>
<td>15</td>
</tr>
</tbody>
</table>
Aggregate operators

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

- count, min, max, average, total

(simplified) syntax:

\[
\text{Function ( } [ \text{distinct} ] \text{ ExpressionOverAttributes } )
\]
Aggregate operators: \texttt{count}

**Syntax:**

- Count the number of tuples:  
  $$\texttt{count} \ (\*)$$

- Count the values in an attribute:  
  $$\texttt{count} \ (\text{Attributo})$$

- Count the \texttt{distinct} values in an attribute:  
  $$\texttt{count} \ (\text{distinct \ Attributo})$$
**count: example and semantics**

*Esempio*: Return the number of children of Franco:

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

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

```sql
select *
from   isFather
where  father = 'Franco'
```
### count: example

#### isFather

<table>
<thead>
<tr>
<th>father</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sergio</td>
<td>Franco</td>
</tr>
<tr>
<td>Luigi</td>
<td>Olga</td>
</tr>
<tr>
<td>Luigi</td>
<td>Filippo</td>
</tr>
<tr>
<td>Franco</td>
<td>Andrea</td>
</tr>
<tr>
<td>Franco</td>
<td>Aldo</td>
</tr>
</tbody>
</table>

**NumChildrenFranco**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
### count and null values

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

Result = 4

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

Result = 3

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

Result = 2

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>27</td>
<td>21</td>
</tr>
<tr>
<td>Aldo</td>
<td>25</td>
<td>NULL</td>
</tr>
<tr>
<td>Maria</td>
<td>55</td>
<td>21</td>
</tr>
<tr>
<td>Anna</td>
<td>50</td>
<td>35</td>
</tr>
</tbody>
</table>
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:

```sql
select avg(income)
from person join isFather on
    name = child
where father = 'Franco'
```
Aggregate operators and null values

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

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>Aldo</td>
<td>25</td>
<td>NULL</td>
</tr>
<tr>
<td>Maria</td>
<td>55</td>
<td>36</td>
</tr>
<tr>
<td>Anna</td>
<td>50</td>
<td>36</td>
</tr>
</tbody>
</table>

The tuple for Andrea is ignored.

<table>
<thead>
<tr>
<th>averageIncome</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
</tr>
</tbody>
</table>
Aggregate operators and target list

The following query does not make sense:

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

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

```sql
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:

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

<table>
<thead>
<tr>
<th>father</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sergio</td>
<td>Franco</td>
</tr>
<tr>
<td>Luigi</td>
<td>Olga</td>
</tr>
<tr>
<td>Luigi</td>
<td>Filippo</td>
</tr>
<tr>
<td>Franco</td>
<td>Andrea</td>
</tr>
<tr>
<td>Franco</td>
<td>Aldo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>father</th>
<th>NumChildren</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sergio</td>
<td>1</td>
</tr>
<tr>
<td>Luigi</td>
<td>2</td>
</tr>
<tr>
<td>Franco</td>
<td>2</td>
</tr>
</tbody>
</table>
Semantics of queries with aggregate operators and grouping

1. execute the query \textit{ignoring the group by clause} and the aggregate operators:

   \begin{verbatim}
   select *
   from isFather
   \end{verbatim}

2. group the \textit{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.

<table>
<thead>
<tr>
<th>person</th>
<th>name</th>
<th>age</th>
<th>income</th>
</tr>
</thead>
</table>
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.

```sql
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 target 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):
  ```sql
  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)):
  ```sql
  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.

```sql
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.

```sql
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)

\[
\text{SelectSQL ::= select AttributeOrExpressionList from TableList [ where SimpleConditions ] [ group by GroupingAttributeList ] [ having AggregationConditions ] [ order by OrderingAttributeList ] [ limit Number ]}
\]
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

```sql
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

<table>
<thead>
<tr>
<th>father</th>
<th>child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sergio</td>
<td>Franco</td>
</tr>
<tr>
<td>Luigi</td>
<td>Olga</td>
</tr>
<tr>
<td>Luigi</td>
<td>Filippo</td>
</tr>
<tr>
<td>Franco</td>
<td>Andrea</td>
</tr>
<tr>
<td>Franco</td>
<td>Aldo</td>
</tr>
<tr>
<td>Luisa</td>
<td>Maria</td>
</tr>
<tr>
<td>Luisa</td>
<td>Luigi</td>
</tr>
<tr>
<td>Anna</td>
<td>Olga</td>
</tr>
<tr>
<td>Anna</td>
<td>Filippo</td>
</tr>
<tr>
<td>Maria</td>
<td>Andrea</td>
</tr>
<tr>
<td>Maria</td>
<td>Aldo</td>
</tr>
</tbody>
</table>
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):

```sql
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:

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

```sql
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

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

is equivalent to

```sql
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:

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

```sql
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)
```
Nested queries: example

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

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

```sql
select name, income
from person
where name in (select father
  from isFather
  where child in (select name
    from person
    where income > 20))
```
Nested queries: \texttt{all} (example)

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

\begin{verbatim}
select name
from person
where income > all ( select income
                       from person
                       where age < 30 )
\end{verbatim}
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.

```sql
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.

```sql
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.

```sql
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 variables 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.
Return the persons having at least a child.

\[
\text{select} \quad * \\
\text{from} \quad \text{person} \\
\text{where} \quad \text{exists} \ (\text{select} \quad * \\
\quad \text{from} \quad \text{isFather} \\
\quad \text{where} \quad \text{father} = \text{name}) \\
\text{or} \\
\text{exists} \ (\text{select} \quad * \\
\quad \text{from} \quad \text{isMother} \\
\quad \text{where} \quad \text{mother} = \text{name})
\]

Attribute \text{name} refers to the relation \text{person} in the \text{from} clause.
Nested queries: visibility

The following query is incorrect:

```sql
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)
```
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

```sql
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

```sql
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.

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

or:

```sql
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:

```sql
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.

\[
\text{create view } \text{ViewName} \left[ (\text{AttributeList}) \right] \text{ as SelectSQL}
\]

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

• Example:

\[
\text{create view adminEmp(Mat,name,surname,Stip) as }
\begin{align*}
\text{select ID, name, surname, salary} \\
\text{from employee} \\
\text{where Depart = 'Administration' and salary > 10}
\end{align*}
\]
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:

```sql
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)
```