

# INTRODUZIONE AD SQL (CAPITOLO 4)

---

R. Basili

a.a. 2021-2022

# SQL

Dopo aver eseguito l'installazione del DBMS e di uno o più database:

	Oracle (7-9i)	MySQL (3.23-4.15)
<b>Accesso al DBMS</b>	<pre>&gt; sqlplus   nome_utente/pwd</pre>	<pre>&gt; mysql -u[nome_user]   -p[pwd] &gt; show databases; &gt; use [nome_database]</pre>
<b>Interpretazione di uno script</b>	<pre>SQL&gt; @nome file</pre>	<pre>&gt; mysql -u[nome_user]   -p[pwd] [nome_db]   &lt;[script.sql]</pre>
<b>Descrizione dello schema di una tabella</b>	<pre>SQL&gt; desc nome_tbl</pre>	<pre>&gt; show create table   [nome_tbl];</pre>
<b>Esecuzione Comando di SO</b>	<pre>SQL&gt;!cmd (Unix/Linux) SQL&gt; \$cmd (Dos)</pre>	

# L'espressione SELECT in SQL

- La struttura base di una interrogazione in SQL e' la seguente:

```
SELECT <Attribute List>  
FROM <Relation List>  
WHERE <Condition>;
```

- Ad es:
  - `SELECT * FROM STUDENTI;`
  - `SELECT age from SAILORS; (*)`
- (\*) SQL non e' case sensitive.

# Select(1) – semantica

```
select [DISTINCT] <target_list>  
from <relation_list>  
[ where <qualification> ]  
[ order by <column(s) [asc\desc]> ]
```

*Semantica*

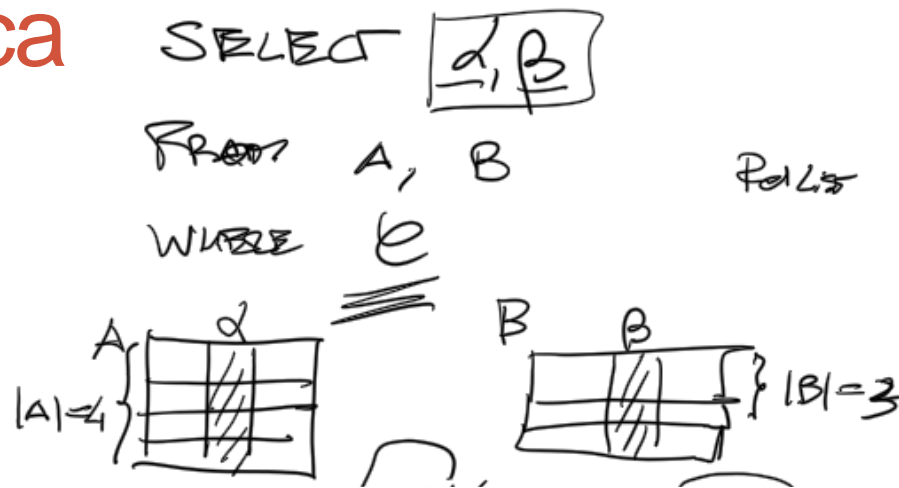


- Eseguire il prodotto cartesiano della `relation_list`
- Scartare le tuple che non soddisfano le condizioni della `qualification`
- Rimuovere gli attributi che non sono presenti nella `target_list`
- Se e' specificata la `DISTINCT` rimuovere le tuple duplicate

# SQL query: semantica

```
select [DISTINCT] <target_list>
from <relation_list>
[ where <qualification> ]
[ order by <column(s)
  [asc\desc]> ]
```

1. Eseguire il prodotto cartesiano della relation\_list
2. Scartare le tuple che non soddisfano le condizioni della qualification
3. Rimuovere gli attributi che non sono presenti nella target\_list



# SQL query: semantica

```
select [DISTINCT] <target_list>
from <relation_list>
[ where <qualification> ]
[ order by <column(s)>
  [asc\desc]> ]
```

1. Eseguire il prodotto cartesiano della relation\_list
2. Scartare le tuple che non soddisfano le condizioni della qualification
3. Rimuovere gli attributi che non sono presenti nella target\_list
4. Se e' specificata la DISTINCT rimuovere le tuple duplicate

SELECT  $\alpha, \beta$   
FROM A, B  
WHERE  $e$

$$\text{card}(\text{sel}_e(\underbrace{\prod_{\alpha, \beta}(A \times B)})) = |\alpha| + |\beta|$$

④ se [DISTINCT]  
opri: tuple per  $\alpha$ .  
 $\alpha_1 \dots \alpha_k \beta_1 \dots \beta_m$   
-----  
si ripete  
non si vengono che una

# WHERE clause

- Es

- `SELECT * FROM STUDENTI WHERE cognome='Rossi'`

- $\sigma_{cognome='Rossi'}(Studenti)$

- `SELECT * FROM STUDENTI WHERE cognome=Rossi`

- `SELECT * FROM STUDENTI  
WHERE (Cognome='Rossi' AND  
          (Nome='Paolo' OR Nome='Mario'))`

- `SELECT * FROM Studenti ORDER BY Cognome`

- `SELECT * FROM Studenti ORDER BY Cognome DESC`

# Operatori

Operatore	Descrizione
=	Uguaglianza
<>	Not Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	Membership in un insieme di valori



# Operatori IN, BETWEEN

- **SELECT \* FROM STUDENTI  
WHERE COGNOME IN ('Rossi', 'Basili')**
- **SELECT \* FROM STUDENTI  
WHERE Cognome NOT IN ('Abi', 'Verdi')**
- **SELECT \* FROM STUDENTI  
WHERE Cognome NOT BETWEEN  
    'Abate' AND 'Babbabei'**

# Alias con Variabili di RANGE

- ```
SELECT Ord_Prodotto.OrderID,  
       Persona.Cogn, Persona.Nome  
FROM Persona,  
       Ord_Prodotto  
WHERE Persona.Cogn='Rosi' AND  
       Persona.Nome='Paolo' AND  
       Ord_Prodotto.CF=Persona.CF
```
- ```
SELECT OP.OrderID, P.Cogn, P.Nome  
FROM Persona AS P, Ord_Prodotto AS OP  
WHERE P.Cogn='Rosi' AND P.Nome='Paolo'  
       AND OP.CF=P.CF
```

# JOIN in SQL

- La query:
  - Trovare tutti i codici dei corsi seguiti da uno studente X Y
- E' espressa dalla seguente relazione

$$\pi_{CID}(\sigma_{X,Y} \text{Studente} \triangleright \triangleleft \text{Segue} )$$

- In SQL

```
SELECT Se.CID
FROM Studente AS St, Segue AS Se
WHERE St.Cogn='X' AND St.Nome='Y'
AND St.StudID=Se.StudID
```

# Tipi di JOIN

- **JOIN:**
  - Restituisce le righe per cui esiste almeno un match in entrambe le tabelle
- **LEFT JOIN:**
  - Restituisce tutte le tuple dalla tabella di sinistra anche se non ci sono match in quella destra
- **RIGHT JOIN:**
  - Restituisce tutte le tuple dalla tabella di destra anche se non ci sono match in quella sinistra

# Operatore UNION

```
Select S.sid AS NUM1, R.bid AS NUM2
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='rossa'
```

## **UNION**

```
Select S.sid AS NUM1, R.bid AS NUM2
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='verde'
```

```
Select S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND
      (B.color='rossa' OR B.color='verde')
```

# Operatore UNION

```
Select R.bid AS NUM1, S.sid AS NUM2  
FROM Sailors S, Boats B, Reserves R  
WHERE S.sid=R.sid AND R.bid=B.bid  
      AND B.color='rossa'
```

## **UNION**

```
Select S.sid AS NUM1, R.bid AS NUM2  
FROM Sailors S, Boats B, Reserves R  
WHERE S.sid=R.sid AND R.bid=B.bid  
      AND B.color='verde'
```

```
Select S.sid  
FROM Sailors S, Boats B, Reserves R  
WHERE S.sid=R.sid AND R.bid=B.bid AND  
(B.color='rossa' OR B.color='verde')
```

# Select- qualification

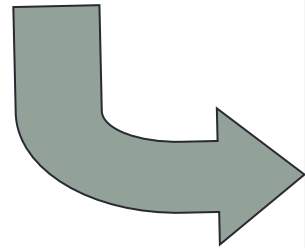
## where ...

### Nested Query:

`<column> [not] in (<query>)`

oppure

**EXISTS** (<query>)



```
Select  S.sname
FROM    Sailors S
WHERE   S.sid IN
        (Select R.sid
         FROM  Reserves R
         WHERE R.bid=103
          AND S.Sid= R.sid)
```

### Null value:

`<column> is [not] null`

### Domain conditions:

`<column> [not] between <lower bound> and <upper bound>`

### String conditions:

`<column> LIKE '<string oppure espr con % o _>'`

# Select- qualification

where

**Nested Query:**

`<column> [not] in (<query>)`

oppure

`EXISTS (<query>)`

```
Select S.sname
FROM Sailors S
WHERE S.sid IN
  (Select R.sid
   FROM Reserves R
   WHERE R.bid=103
   AND S.Sid = R.sid)
```

```
Select S.sname
FROM Sailors S
WHERE EXISTS
  (Select R.sid
   FROM Reserves R
   WHERE R.bid=103
   AND S.Sid = R.sid)
```

```
Select S.sname
FROM Sailors S
WHERE S.sid IN
  (Select R.sid
   FROM Reserves R
   WHERE R.bid=103 )
```

```
Select S.sname
FROM Sailors S,
      Reserves R
WHERE S.sid=R.sid
AND
      R.bid=103
```



- WHERE age IS NOT NULL
  - %il valore della colonna «age» non può essere NULL
- WHERE age IS NULL
  - %il valore della colonna «age» DEVE essere NULL
- *<column> LIKE '<string oppure espr con % o \_>'*
- WHERE Surname LIKE 'BAS\*'
  - SURNAME = 'BASili' TRUE
  - SURNAME = 'BAS' TRUE
  - SURNAME = 'Obama' FALSE

## Q1 e Q2

 Trova il nome dei marinai (tabella S) che hanno riservato la barca numero 103.

```
SELECT S.sname
FROM Sailors S,Reserves R
WHERE S.sid=R.sid AND R.bid=103
```

 Trova il nome dei marinai (tabella S) che hanno riservato una barca rossa.

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid=R.sid AND R.bid=B.bid
AND B.color='rossa'
```

## Q4 e Q5



*Trova il nome dei marinai (tabella S) che hanno riservato una barca rossa oppure una barca verde.*

```
SELECT S.sname
FROM Sailors S, Reserves R, B Boats
WHERE S.sid=R.sid AND R.bid = B.bid
AND (B.color='rossa' OR B.color='verde')
```



*Trova il nome dei marinai (tabella S) che hanno riservato una barca rossa ed una barca verde.*

```
SELECT S.sid, S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='rossa'
INTERSECT
SELECT S2.sid, S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid=R2.sid AND R2.bid=B2.bid
AND B2.color='verde'
```

Q9



Trova il nome dei marinai (tabella S) che hanno riservato **tutte** le barche

$$M/N = \pi_{\beta}(M) - \pi_{\beta}((\pi_{\beta}(M) \times N) - M)$$

$$S/R = \pi_{\beta, \text{sname}}(S) - \pi_{\beta, \text{sname}}((\pi_{\beta}(S) \times B) - S)$$



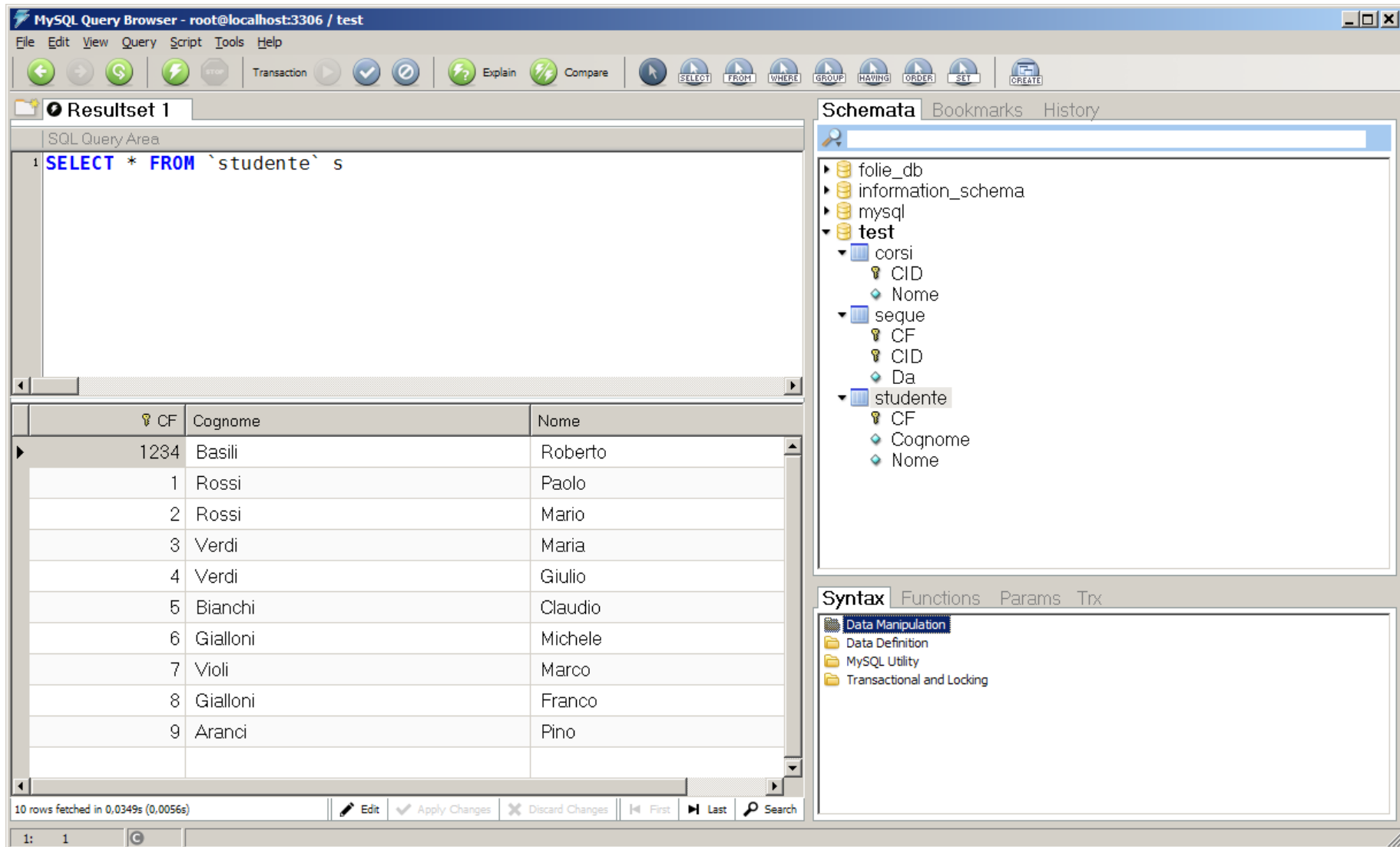
# Esempio (da ER precedenti)

- Due entità: Studenti e Corsi
- Una relazione: Segue
- Studio di: JOIN, LEFT e RIGHT (OUTER) JOIN, Nested queries, Divisione

# ENTITA': STUDENTE

```
-- Table structure for table `studente`  
CREATE TABLE `studente` (  
  `CF` int(10) unsigned NOT NULL auto_increment,  
  `Cognome` varchar(45) NOT NULL,  
  `Nome` varchar(45) NOT NULL,  
  PRIMARY KEY (`CF`)  
);  
-- Data for table `studente`  
INSERT INTO `studente` VALUES  
  (1234, 'Basili', 'Roberto'), (1, 'Rossi', 'Paolo'), (2, 'Rossi', 'Mario'), (3, 'Verdi', 'Maria'), (4, 'Verdi', 'Giulio'), (5, 'Bianchi', 'Claudio'), (6, 'Gialloni', 'Michele'), (7, 'Violi', 'Marco');
```

# MySQL Query Browser



The screenshot displays the MySQL Query Browser interface. The main window shows a query result set for the query `SELECT * FROM `studente` s`. The result set contains 10 rows, with the first row highlighted. The columns are labeled 'CF', 'Cognome', and 'Nome'. The status bar at the bottom indicates that 10 rows were fetched in 0.0349s (0.0056s).

The right-hand pane shows the 'Schemata' view, displaying a tree structure of the database schema. The 'test' database is expanded, showing the following tables and their columns:

- `corsi`: CF, Nome
- `seque`: CF, CID, Da
- `studente`: CF, Cognome, Nome

The 'Syntax' pane at the bottom right shows a tree structure of SQL syntax categories, including 'Data Manipulation', 'Data Definition', 'MySQL Utility', and 'Transactional and Locking'.

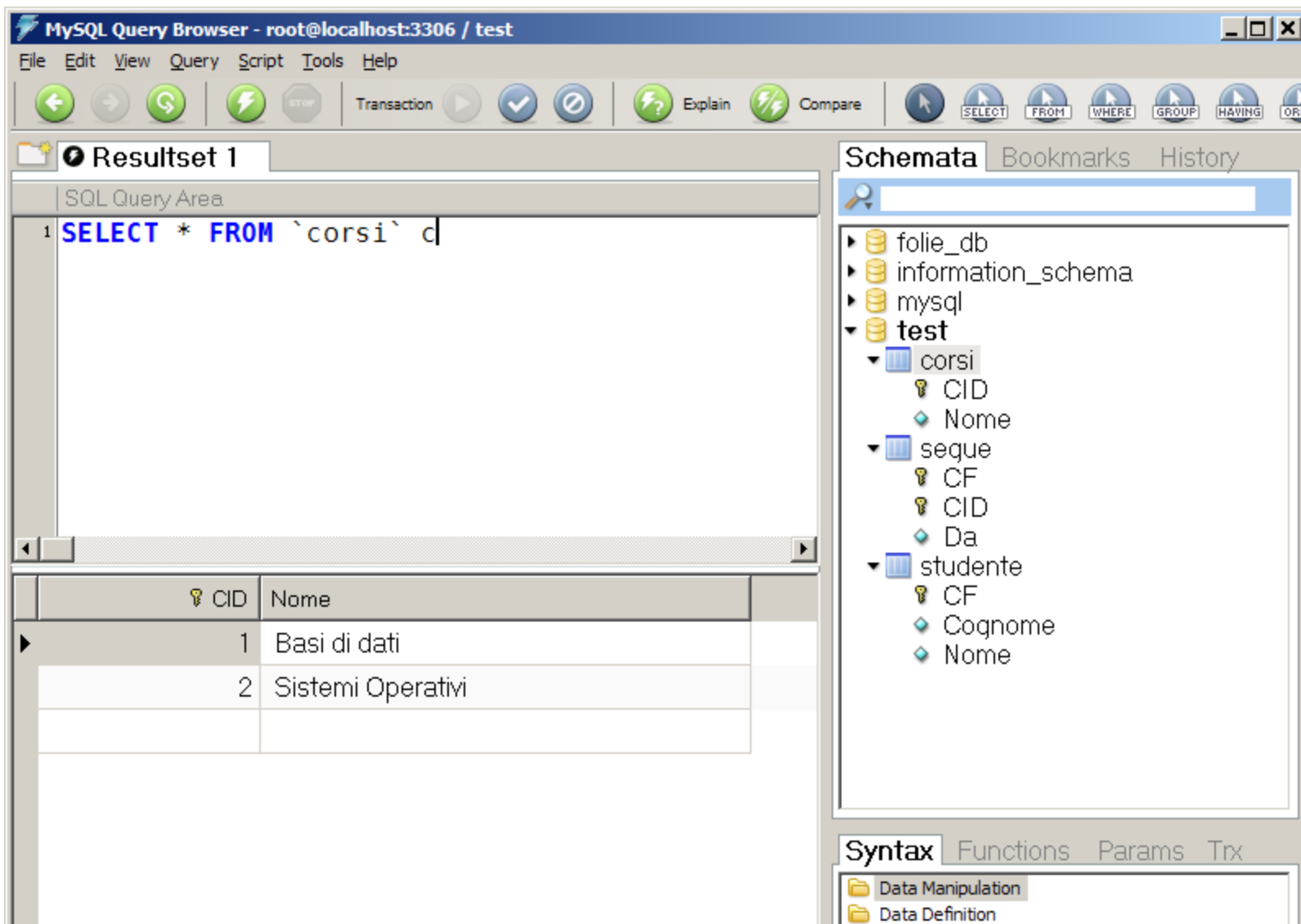
CF	Cognome	Nome
1234	Basili	Roberto
1	Rossi	Paolo
2	Rossi	Mario
3	Verdi	Maria
4	Verdi	Giulio
5	Bianchi	Claudio
6	Gialloni	Michele
7	Violi	Marco
8	Gialloni	Franco
9	Aranci	Pino



# ENTITA': Corsi

```
--  
DROP TABLE IF EXISTS `corsi`;  
CREATE TABLE `corsi` (  
  `CID` int(10) unsigned NOT NULL auto_increment,  
  `Nome` varchar(45) NOT NULL,  
  PRIMARY KEY (`CID`)  
);  
  
INSERT INTO `corsi` VALUES (1,'Basi di  
dati'), (2,'Sistemi Operativi');
```

# Istanza di “Corsi”



The screenshot shows the MySQL Query Browser interface. The main window displays the SQL Query Area with the query: `SELECT * FROM `corsi` d`. Below the query area, the Resultset 1 is shown as a table with two columns: CID and Nome. The table contains two rows of data.

CID	Nome
1	Basi di dati
2	Sistemi Operativi

The right-hand side of the interface shows the Schemata panel, which lists the databases and tables. The 'test' database is expanded, showing the 'corsi' table and its columns: CID, Nome, seque, CF, CID, Da, studente, CF, Cognome, and Nome.

The bottom of the interface shows the Syntax panel, which lists the categories: Data Manipulation and Data Definition.

# Relazione: Segue

```
-- -- Table structure for table `segue`

CREATE TABLE `segue` (
  `CF` int(10) unsigned NOT NULL auto_increment,
  `CID` int(10) unsigned NOT NULL,
  `Da` datetime NOT NULL,
  PRIMARY KEY (`CF`,`CID`),
  FOREIGN KEY `CF` REFERENCES Students(`CF`),
  FOREIGN KEY `CID` REFERENCES CORSI(`CID`),
);

-- Dumping data for table `segue`
INSERT INTO `segue` VALUES
  (1,1,'0001-01-08'),(2,1,'2001-01-08'),
  (3,1,'2001-01-20 '), (4,1,'2001-10-08'),
  (1234,2,'2001-01-08 '), (1,2,'2001-01-00'),
  (1234,1,'2001-01-09');
```

# MySQL Query Browser

The screenshot shows the MySQL Query Browser interface. The title bar reads "MySQL Query Browser - root@localhost:3306 / test". The menu bar includes "File", "Edit", "View", "Query", "Script", "Tools", and "Help". The toolbar contains various icons for navigation and execution, including "Transaction", "Explain", "Compare", and SQL keywords like "SELECT", "FROM", "WHERE", "GROUP", "HAVING", and "ORDER".

The main window is titled "Resultset 1" and contains an "SQL Query Area" with the following query:

```
1 SELECT * FROM `segue` s|
```

Below the query area, a table displays the results of the query. The table has three columns: "CF", "CID", and "Da". The data rows are as follows:

CF	CID	Da
1	1	0001-01-08...
2	1	2001-01-08...
3	1	2001-01-20...
4	1	2001-10-08...
1234	2	2001-01-08...
1	2	2001-01-00...
1234	1	2001-01-09...

On the right side, the "Schemata" panel shows a tree view of the database structure. The "test" database is expanded, showing the following tables and their columns:

- corsi
  - CID
  - Nome
- segue
  - CF
  - CID
  - Da
- studente
  - CF
  - Cognome
  - Nome

At the bottom, the "Syntax" panel is visible, showing a list of SQL categories: "Data Manipulation", "Data Definition", "MySQL Utility", and "Transactional and Locking".

# Interrogazioni: JOIN

```
--- QUERY:  
/* JOIN senza operatore */  
SELECT Se.CID, St.CF, St.Cognome, St.Nome  
FROM Studente AS St, Segue AS Se  
WHERE (St.Cognome='Rossi' AND  
St.CF=Se.CF) ;
```

## Interrogazioni: JOIN (2)

```
/* operatore JOIN con condizione */  
SELECT *  
FROM Studente AS St  
JOIN Segue AS Se ON St.CF=Se.CF  
WHERE St.Cognome='Rossi';
```

# Interrogazioni: OUTER JOIN

```
/* Full JOIN, RIGHT JOIN e LEFT JOIN */  
SELECT Se.CID, ST.CF, ST.Cognome, St.Nome  
FROM Studente AS St  
JOIN Segue AS Se ON Se.CF=St.CF;
```

```
SELECT Se.CID, ST.CF, ST.Cognome, St.Nome  
FROM Studente AS St  
LEFT JOIN Segue AS Se ON Se.CF=St.CF;
```

# Join

MySQL Query Browser - root@localhost:3306 / test

File Edit View Query Script Tools Help

Transaction Explain Compare SELECT FROM WHERE GROUP HAVING ORDER SET CREATE

Resultset 1 Resultset 2 **Resultset 3** Resultset 4

SQL Query Area

```

1 /* Full JOIN, RIGHT JOIN e LEFT JOIN */
2 SELECT Se.CID, ST.CF, ST.Cognome, St.Nome
3 FROM Studente AS St INNER JOIN Segue AS Se ON Se.CF=St.CF;

```

CID	CF	Cognome	Nome
1	1	Rossi	Paolo
2	1	Rossi	Paolo
1	2	Rossi	Mario
1	3	Verdi	Maria
1	4	Verdi	Giulio
1	1234	Basili	Roberto
2	1234	Basili	Roberto

7 rows fetched in 0,0190s (0,0067s) | Edit | Apply Changes | Discard Changes | First | Last | Search

Schemata Bookmarks History

- folie\_db
- information\_schema
- mysql
- test**
  - corsi
    - CID
    - Nome
  - segue
    - CF
    - CID
    - Da
  - studente
    - CF
    - Cognome

Syntax Functions Params Trx

- Data Manipulation
  - DELETE
  - DO
  - HANDLER
  - INSERT
  - LOAD DATA INFILE
  - REPLACE
  - SELECT**
  - Subquery
  - TRUNCATE
  - UPDATE



SQL Query Area

```
1 /* Full JOIN, RIGHT JOIN e LEFT JOIN */
2 SELECT Se.CID, ST.CF, ST.Cognome, St.Nome
3 FROM Studente AS St LEFT JOIN Segue AS Se ON Se.CF=St.CF;
```

Schemata Bookmarks History

- folie\_db
- information\_schema
- mysql
- test**
  - corsi

CID	CF	Cognome	Nome
2	1234	Basili	Roberto
1	1	Rossi	Paolo
2	1	Rossi	Paolo
1	2	Rossi	Mario
1	3	Verdi	Maria
1	4	Verdi	Giulio
NULL	5	Bianchi	Claudio
NULL	6	Gialloni	Michele
NULL	7	Violi	Marco

9 rows fetched in 0,0130s (0,0014s) | Edit | Apply Changes | Discard Changes | First

Result Grid Filter Rows:

CID	CF	Cognome	Nome
1	1234	Basili	Roberto
2	1234	Basili	Roberto
1	1	Rossi	Paolo
2	1	Rossi	Paolo
1	2	Rossi	Mario
1	3	Verdi	Maria
1	4	Verdi	Giulio
NULL	5	Bianchi	Claudio
NULL	6	Gialloni	Michele
NULL	7	Violi	Marco

# Interrogazioni: JOIN MULTIPLE

```
/* Multiple JOIN */
```

```
SELECT Se.CID, Co.Nome, ST.CF, ST.Cognome, St.Nome
```

```
FROM Studente AS St
```

```
left JOIN Segue AS Se ON Se.CF=St.CF
```

```
left JOIN Corsi As Co ON Co.CID = Se.CID;
```

```
SELECT Se.CID, Co.Nome, ST.CF, ST.Cognome, St.Nome
```

```
FROM Studente AS St
```

```
JOIN Segue AS Se ON Se.CF=St.CF
```

```
left JOIN Corsi As Co ON Co.CID = Se.CID;
```

# Interrogazioni: QUERY ANNIDATE

```
/* Query ANNIDATE: Uso dell'operatore IN */  
Select  St.Nome, St.Cognome, St.CF  
FROM    Studente St  
WHERE   St.CF IN (  
        Select Se.CF  
        FROM   Segue Se, Corsi Co  
        WHERE  Co.Nome='Basi di Dati'  
              AND Se.CID=Co.CID );
```

```
/* Query ANNIDATE: Uso dell'operatore IN */  
Select  St.Nome, St.Cognome, St.CF  
FROM Studente St  
INTERESECT  
Select Se.CF  
      FROM Segue Se, Corsi Co  
WHERE Co.Nome='Basi di Dati'  
      AND Se.CID=Co.CID;
```

# Operazione di Divisione

$$A/B = \pi_x(A) - \pi_x((\pi_x(A) \times B) - A)$$

Trova i marinai che hanno prenotato **tutte** le barche

$$\text{Pren}/B = \{ \langle x \rangle \mid \exists \langle x, y \rangle \in A \quad \forall \langle y \rangle \in B \}$$

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s3	p2

Pren

pno
p2

B1

pno
p2
p4

B2

Pren/B1



sno
s1
s3

Pren/B2



sno
s1

# Interrogazioni: La divisione in SQL

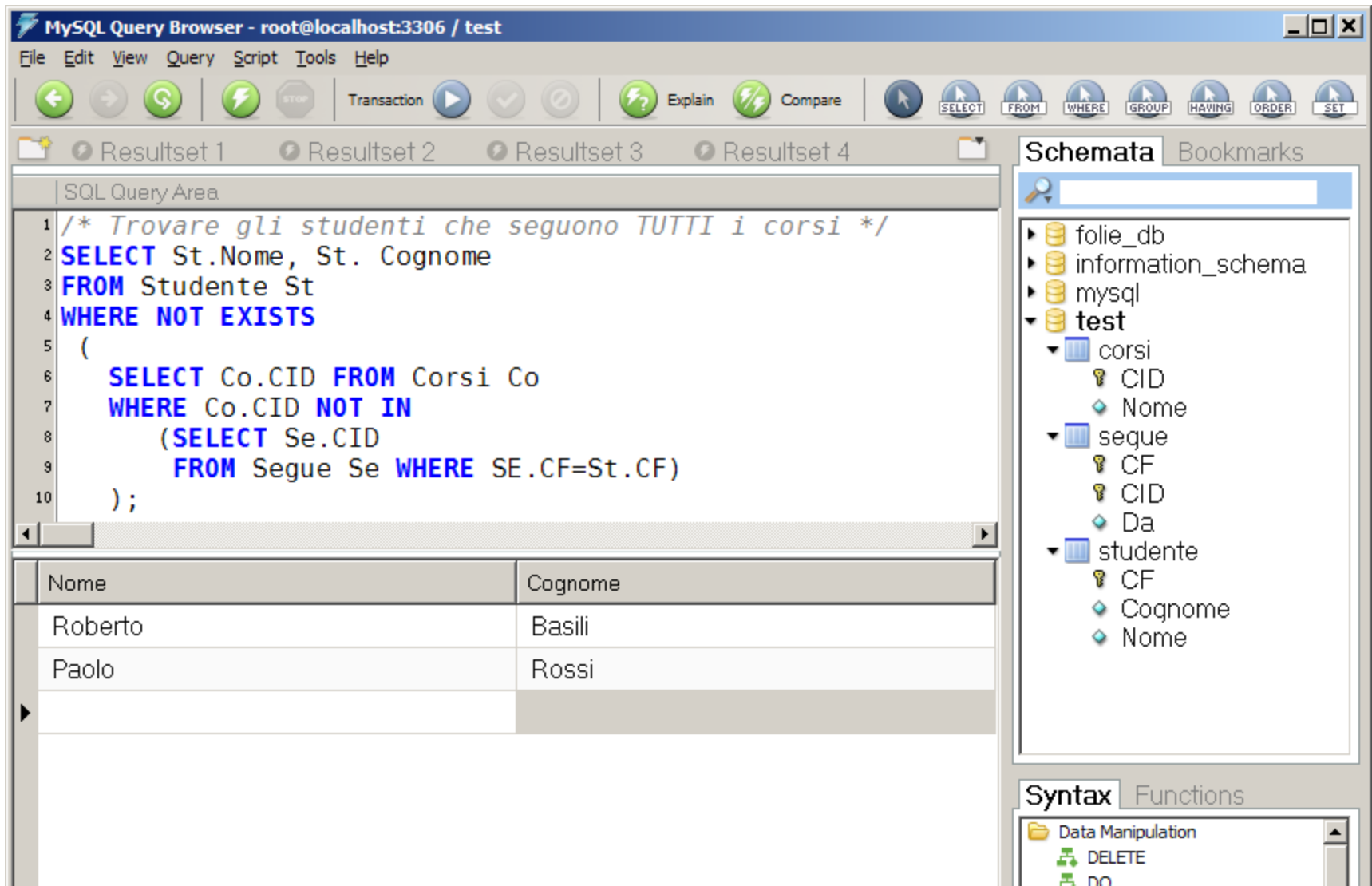
```

/* Trovare gli studenti che seguono TUTTI i corsi */
SELECT St.Nome, St. Cognome
FROM Studente St
WHERE NOT EXISTS
(
  SELECT Co.CID FROM Corsi Co
  WHERE Co.CID NOT IN
    (SELECT Se.CID
     FROM Segue Se WHERE SE.CF=St.CF)
);

```

$$= \pi_{SID}(Se) - \pi_{SID}((\pi_{SID}(Se) \times Co) - Se)$$

# Divisione



The screenshot shows the MySQL Query Browser interface. The main window displays a SQL query in the 'SQL Query Area' and the resulting data in a table below it. The query is designed to find students who are enrolled in all courses.

```
1 /* Trovare gli studenti che seguono TUTTI i corsi */
2 SELECT St.Nome, St. Cognome
3 FROM Studente St
4 WHERE NOT EXISTS
5 (
6     SELECT Co.CID FROM Corsi Co
7     WHERE Co.CID NOT IN
8         (SELECT Se.CID
9          FROM Segue Se WHERE SE.CF=St.CF)
10 );
```

Nome	Cognome
Roberto	Basili
Paolo	Rossi

The right-hand side of the interface shows the 'Schemata' panel, which lists the databases and tables in the current environment. The 'test' database is expanded, showing tables like 'corsi', 'seque', and 'studente' with their respective columns.

Below the 'Schemata' panel, the 'Syntax' panel is visible, showing a list of SQL functions under the 'Data Manipulation' category, including 'DELETE' and 'DO'.

# Operatori di Aggregazione

```
count (* | [DISTINCT] Attr)
```

```
sum ([DISTINCT] Attr)
```

```
AVG ([DISTINCT] Attr)
```

```
MIN (A)
```

```
MAX (A)
```

non esprimibili  
in algebra  
relazionale!

1. `count (* | [DISTINCT] Attr)` restituisce il numero di righe `count(*)`, con `Distinct` restituisce il numero di distinti valori di `Attr`
2. `sum ([DISTINCT] Attr)` somma dei valori dell'attributo `Attr` su tutte le tuple della rel.
3. `avg ([DISTINCT] Attr)` media dei valori dell'attributo `Attr` su tutte le tuple
4. `MAX (Attr)` massimo valore di `Attr` su tutte le tuple
5. `MIN (Attr)` minimo valore di `Attr` su tutte le tuple



# Query con operazioni di aggregazione

 Trova l'età massima dei marinai che hanno rating uguale a 10

```
SELECT MAX(S.age)
FROM Sailors S
WHERE S.rating=10
```

 Trova il nome e l'età del marinaio più vecchio

```
SELECT S.sname, MAX(S.age)
FROM Sailors S
```

**Query ILLEGALE!**

◆ Quando in un argomento della select compaiono delle funzioni di aggregazione, allora non possono comparire **espressioni a livello di riga**, come ad esempio il nome di un attributo

```
SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
      FROM Sailors S2) = S.age
```

# Clausole di GROUP BY ed HAVING

```
SELECT [DISTINCT]
    <target_list>
FROM <relation_list>
WHERE <qualification> ]
GROUP BY <grouping-list>
HAVING <group-qualification>
```

- ◆ Se non vogliamo applicare l'operazione di aggregazione a tutte le tuple che soddisfano la condizione di **where**, possiamo applicarla a sottoinsiemi (gruppi) di righe distinti in base al valore degli attributi presenti nella *grouping-list* della clausola **GROUP BY**.
- ◆ Possiamo inoltre imporre delle condizioni di selezione su questi gruppi specificandole nella clausola **HAVING**.

# Valutazione Concettuale

```
SELECT [DISTINCT] <target_list>  
FROM <relation_list>  
WHERE <qualification> ]  
GROUP BY <grouping-list>  
HAVING <group-qualification>
```

**Solamente**  
attributi presenti nella  
group by oppure come  
argomenti di operazioni  
di aggregazione

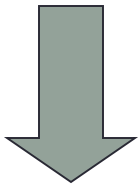
1. Viene calcolato il prodotto cartesiano delle tabelle presenti nella *relation\_list*, vengono scartate le tuple che non soddisfano le condizioni di *qualification*.
2. Divido secondo la *grouping-list* la relazione ottenuta dalla fase 1 in gruppi di tuple basandomi sulla *grouping-list*.
3. Applico la *group-qualification* per eliminare alcuni gruppi ed ottengo una tupla (risposta) per ogni gruppo

 Per ogni barca rossa, trova il numero delle prenotazioni

```
SELECT B.bid,COUNT(*) AS numeroprenot
FROM Boats B, Reservations R
WHERE R.bid=B.bid
GROUP BY B.bid,
HAVING B.color='rossa'
```


Query **ILLEGALE!**

**E' rifiutata anche se il colore della barca  
e' unico per ogni gruppo (B.bid e' chiave)**




```
SELECT B.bid, COUNT(*) AS numeroprenot
FROM Boats B, Reservations R
WHERE R.bid=B.bid AND B.color='rossa'
GROUP BY B.bid
```

Query **CORRETTA**

 Trova l'età del più giovane marinaio maggiorenne per ogni livello di rating con almeno due di tali marinai

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```

 Trova l'età del marinaio maggiorenne più giovane per ogni livello di rating con almeno due marinai (di ogni età)

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING 1 < (SELECT COUNT(*) FROM Sailors S2
            WHERE S.rating=S2.rating)
```



*Trova l'età del marinaio più giovane per ogni livello di rating con almeno due marinai maggiorenni*

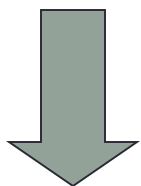
```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
GROUP BY S.rating
HAVING 1 < (SELECT COUNT (*)
            FROM Sailors S2
            WHERE (S2.age>18)
                  AND
                  (S.rating=S2.rating))
```

 Trova i rating per i quali l'età media e' la minima su tutti i rating

```
SELECT S.rating
FROM Sailors S
WHERE S.age=(SELECT MIN (AVG (S2.age)) FROM Sailors S2)
```

**Query ILLEGALE!**

**Le operazioni di aggregazione non possono essere annidate**



**Soluzione alternativa**

```
SELECT Temp.rating, Temp.media
FROM (SELECT S.rating, AVG(S.age) AS media
      FROM Sailors S
      GROUP BY S.rating) AS Temp
WHERE Temp.media=(SELECT MIN(Temp.media) FROM Temp)
```

**Query CORRETTA**

# Valori Nulli

In SQL si utilizza un valore chiamato *null* per indicare che il valore di un determinato campo e' non definito.

E' necessario definire una *logica a tre valori* (*true, false e null*) per definire il comportamento di alcune operazioni quando queste vengono applicate a campi *null*

- ◆ Operatori di confronto (es:Attr>8)
- ◆ Operatori logici (***AND, OR e NOT***)

La clausola ***WHERE*** deve essere rianalizzata alla luce dei possibili valori *null*

- ◆ Deve essere definito un nuovo operatore

*IS [NOT] NULL*

per verificare quando un campo [non] ha valore *null*.



# Considerazioni sul *null*

- La clausola **WHERE** fallisce su valori sconosciuto (in presenza di **NULL**) e le corrispondenti tuple vengono scartate
- **COUNT (\*)** conta ANCHE i valori **NULL**
- **SUM ()** e le altre op. di aggregazione agiscono sul sottoinsieme dei valori non **NULL**

## Valori Nulli ed OUTER JOIN

La logica a tre valori (*true, false e null*) ci porta a definire un nuovo operatore l'OUTER JOIN

Considerando il join di due tabelle  $S \triangleright \triangleleft_C R$ , aggiungo nel risultato anche le tuple di  $S$  che non hanno corrispettivo in  $R$ , gli attributi di  $R$  vengono ugualmente inseriti nel risultato ma con valori *null*.

```
SELECT S.sid, R.bid
FROM Sailors S NATURAL LEFT OUTER JOIN Reserves R
```

sid	sname	rating	age
22	Davide	7	45.0
31	Luca	8	55.5
58	Remo	10	35.0

Sailors

sid	bid	day
22	101	10/10/03
58	103	11/12/02

Reserves



sid	bid
22	101
31	<i>null</i>
58	103

# Set queries

- Con un comando di SELECT non si può rappresentare l'unione
- Syntax:  
*SelectSQL* { < union | intersect | except > [ all ] *SelectSQL* }
- Trova i nomi ed i cognomi degli impiegati:  

```
select FirstName as Name
from Employee
union
select Surname
from Employee
```
- OSS: I duplicati sono rimossi (per definizione di insieme) (a meno che non sia usata la opzione `all`)

# Tuple constructor

- The comparison with the nested query may involve more than one attribute
- The attributes must be enclosed within a pair of curved brackets (tuple constructor)
- The previous query can be expressed in this way:

```
select *
from Person P
where (FirstName, Surname) not in
      (select P1.Firstname AS FirstName1,
             P1.Surname AS Surname1
       from Person P1
       where P1.TaxCode <> P.TaxCode)
```

# Scope of variables

- Incorrect query:

```
select *
from Employee
where Dept in
    (select DeptName
     from Department D1
     where DeptName = 'Production')

or
    Dept in (select DeptName
            from Department D2
            where D2.City = D1.City)
```

- The query is incorrect because variable D1 is not visible in the second nested query

# Views, 1

- Syntax:

```
create view ViewName [ (AttributeList) ] as SelectSQL  
  [ with [ local | cascaded ] check option ]
```

```
create view AdminEmployee  
  (RegNo, FirstName, Surname, Salary) as  
select RegNo, FirstName, Surname, Salary  
from Employee  
where Dept = 'Administration' and Salary > 10
```

```
create view JuniorAdminEmployee as  
select *  
from AdminEmployee  
where Salary < 50  
with check option
```

# Views, 2

- SQL views cannot be mutually dependent (no recursion)
- The `check option` operates when a view content is updated
- Views can be used to formulate complex queries
  - Views decompose the problem and produce a more readable solution
- Views are sometimes necessary to express certain queries:
  - queries that combine and nest several aggregate operators
  - queries that make a sophisticated use of the union operator

# Views and queries, 1

- Find the department with the highest salary expenditure (without a view):

```
select Dept
from Employee
group by Dept
having sum(Salary) >= all
      (select sum(Salary)
       from Employee
       group by Dept)
```

- This solution may not be recognized by all SQL systems



# Views and queries, 2

- Find the department with the highest salary expenditure (using a view):

```
create view SalaryBudget (Dept, SalaryTotal)
as
```

```
select Dept, sum(Salary)
from Employee
group by Dept
```

```
select Dept
from SalaryBudget
where SalaryTotal = (select max(SalaryTotal)
                    from SalaryBudget)
```

# Views and queries

- Find the average number of offices per department:
  - Incorrect solution (SQL does not allow a cascade of aggregate operators):

```
select avg(count(distinct Office))
from Employee
group by Dept
```

- Correct solution (using a view):

```
create view DeptOff(Dept, NoOfOffices) as
select Dept, count(distinct Office)
from Employee
group by Dept
```

```
select avg(NoOfOffices)
from DeptOffice
```

# Access control

- Every component of the schema can be protected (tables, attributes, views, domains, etc.)
- The owner of a resource (the creator) assigns privileges to the other users
- A predefined user `_system` represents the database administrator and has complete access to all the resources
- A privilege is characterized by:
  - the resource
  - the user who grants the privilege
  - the user who receives the privilege
  - the action that is allowed on the resource
  - whether or not the privilege can be passed on to other users

# Types of privilege

- SQL offers six types of privilege
  - `insert`: to insert a new object into the resource
  - `update`: to modify the resource content
  - `delete`: to remove an object from the resource
  - `select`: to access the resource content in a query
  - `references`: to build a referential integrity constraint with the resource (may limit the ability to modify the resource)
  - `usage`: to use the resource in a schema definition (e.g., a domain)

# grant **and** revoke

- To grant a privilege to a user:

```
grant < Privileges | all privileges > on Resource  
to Users [with grant option]
```

- grant option specifies whether the privilege of propagating the privilege to other users must be granted

- E.g.:

```
grant select on Department to Stefano
```

- To take away privileges:

```
revoke Privileges on Resource from Users  
[ restrict | cascade ]
```

# Generic integrity constraints

- The `check` clause can be used to express arbitrary constraints during schema definition
- Syntax:
  - `check` (*Condition*)
- *Condition* is what can appear in a `where` clause (including nested queries)
- E.g., the definition of an attribute `Superior` in the schema of table `EMPLOYEE`:

```
Superior character(6)
check (RegNo like "1%" or
      Dept = (select Dept
              from Employee E
              where E.RegNo = Superior))
```

# Assertions

- Assertions permit the definition of constraints outside of table definitions
- Useful in many situations (e.g., to express generic inter-relational constraints)
- An assertion associates a name to a `check` clause;  
syntax:

```
create assertion AssertionName check (Condition)
```

- There must always be at least one tuple in table `EMPLOYEE`:

```
create assertion AlwaysOneEmployee  
  check (1 <= (select count(*)  
              from Employee))
```

# Table Constraints

- Quando e' necessario specificare dei constraint complessi si utilizza la clausola CHECK <condizione>

```
CREATE TABLE Sailors ( sid      INTEGER,
                        sname   CHAR(10),
                        rating  INTEGER,
                        age     REAL,
                        PRIMARY KEY (sid),
                        CHECK (rating>=1 AND <=10) )
```

*oppure:*

```
CREATE DOMAIN ratingVal INTEGER DEFAULT 1
        CHECK ( VALUE>=1 AND VALUE<=10)
```



# DOMINI

- E' possibile definire dei domini di valori dedicati dipendenti da specifico dominio applicativo utilizzando la clausola `CHECK <Condizione>`


```
CREATE DOMAIN ValoriEx INTEGER DEFAULT 1  
CHECK (VALUE >=1 AND VALUE <=10)
```


*Oss: Expertise ha tipo INTEGER e domini diversi di tipo INTEGER sono SEMPRE compatibili*  
*Alternativa:*

```
CREATE TYPE Expertise AS INTEGER  
CREATE DOMAIN ValExp1 Expertise ...  
CREATE DOMAIN ValExp2 Expertise ...  
CREATE DOMAIN SAILOR_AGE INTEGER ...
```

## Assertzioni: Integrity Constraints su piu' tabelle

- L'asserzione e' un IC definito separatamente dallo schema delle tabelle a cui fa riferimento e di cui definisce i vincoli di integrita`

 `CREATE ASSERTION SmallClub  
CHECK ((SELECT COUNT(S.sid) FROM Sailors S)  
+ (SELECT COUNT(B.bid) FROM Boats B) < 100)`

 `CREATE TABLE Sailors (sid INTEGER, sname CHAR(10),  
rating INTEGER, age REAL  
PRIMARY KEY (sid),  
CHECK(rating >= 1 AND rating <= 10)  
CHECK((SELECT COUNT(S.sid) FROM Sailors S) +  
(SELECT COUNT(B.bid) FROM Boats B) < 100))`

# I Trigger

- Un **trigger** è una procedura che viene attivata automaticamente a seguito di cambiamenti sul DBMS. Un database in cui sono stati definiti dei trigger si dice **ATTIVO**
- I trigger seguono il paradigma **ECA**:
  - **E**vento che attiva il trigger
  - **C**ondizione che deve essere soddisfatta affinché il trigger venga eseguito
  - **A**zione: le operazioni che esegue il trigger

Può essere visto come uno script *daemon* che monitorizza il database.

## Esempio di trigger

**EVENTO:** Update della tabella Studenti

**CONDIZIONE:** controllo età del nuovo studente

**AZIONE:** gestione del contatore dei maggiorenni

```
CREATE TRIGGER init_cont  
BEFORE INSERT ON Studenti  
DECLARE Contatore INTEGER  
BEGIN Contatore:=0 END
```

**Evento**

**Azione**

```
CREATE TRIGGER incr_cont  
AFTER INSERT ON Studenti  
WHEN (new.età>18)  
FOR EACH ROW  
BEGIN Contatore:=Contatore+1 END
```

**Evento**

**Condizione**

**Azione**

# Esempio di trigger

**EVENTO:** Update della quota disponibile in magazzino

**CONDIZIONE:** controllo se la quantità disponibile e' inferiore alla quantità di riordino

**AZIONE:** emissione di un ordine altrimenti eccezione

```
CREATE TRIGGER gestione_magazzino
AFTER UPDATE OF qta-disp ON Magazzino
WHEN (NEW.qta-disp < NEW.qta-riord)
FOR EACH ROW
X EXCEPTION
BEGIN
IF NEW.QTA-DISP<0 THEN RAISE (X) ;
INSERT INTO RIORDINO
VALUES (NEW.COD-PROD, SYSDATE,
                NEW.QTA-RIORD)
END
```

**Evento**  
**Condizione**

**Azione**