# Data Management

**Maurizio Lenzerini**

*Dipartimento di Informatica e Sistemistica "Antonio Ruberti"*
*Sapienza Università di Roma*

**Academic year 2024/2025**

*Part 1*
*Introduction to the course and recap of the relational data model*

**http://www.dis.uniroma1.it/~lenzerini/home/?q=node/53**

# This course is for …

- Students of the Master of Science in Engineering in Computer Science and Master in Ingegneria Gestionale
  - 6 credits
- Prerequisites
  - A good knowledge of the fundamentals of Programming Structures and Programming Languages
  - A good knowledge of the fundamentals of Databases, in particular SQL, relational data model, Entity-Relationship data model, conceptual and logical database design

# Objectives

- *Knowledge on the structure and the functionalities of Data Management systems from the point of view of data administrators*
- *Knowledge on the structure and the functionalities of Data Management systems from the point of view of Data Management tool designers*
- *Other topics in data management, such as data warehousing, noSQL, etc.*

# Organization of the course

Teacher: Maurizio Lenzerini

Home page of Prof. Maurizio Lenzerini

http://www.diag.uniroma1.it/lenzerini

Home page of the course:

http://www.diag.uniroma1.it/lenzerini/home/?q=node/53

Office hours:

- Tuesday, 5:00 pm
- Meet room at https://meet.google.com/hzy-save-oqw

# Organization of the course

- Lectures (via Eudossiana 18 and ON-LINE at https://uniroma1.zoom.us/j/83220889311?pwd=ZkxrQ3crNFJDVlIwS21jelp4bjRXZz09):
  - Monday, 01:00 pm – 03:00 pm (Classroom 41)
  - Wednesday, 10:00 am – 01:00 pm (Classroom 41)
- Exercises during lectures
- Possible individual or group projects
- Exam
  - written exam
  - oral exam (if needed)

# Organization of the course

Material

- M. Lenzerini, Lecture notes, Available for download from the Moodle system
- R. Ramakrishnan, J. Gehrke. Database Management Systems. McGraw-Hill
- Papers on specific topics

- More material available in the course web site
  - exercises
  - problems proposed in past exams

# Course topics

- Data Warehousing
  - Data warehouse architectures and operators
  - Data warehouse design
- NoSQL databases
  - Document-based databases
  - Graph databases OLAP vs OLTP
- The modules of a Data Base Management System (DBMS)
  - The structure of a DBMS
  - Buffer manager
- Transaction management
  - The concept of transaction
  - Concurrency management
- Crash management
  - Classification of failures
  - Recovery
- Physical structures for data bases
  - File organizations for data base management
  - Principles of physical database design
- Query processing
  - Evaluation of relational algebra operators
  - Fundamentals of query optimization

# Recap on relational data bases

# The Relational Data Model (E.F. Codd – 1970)

## CHECKING-ACCOUNT Table

| branch-name | account-no | customer-name | balance |
|---|---|---|---|
| Orsay | 10991-06284 | Abiteboul | $3,567.53 |
| Hawthorne | 10992-35671 | Hull | $11,245.75 |
| … | … | … | … |

- The Relational Data Model uses the mathematical concept of a relation as the formalism for describing and representing data.
- Question: What is a relation?
- Answer:
  - Mathematically speaking, a relation is a subset of a cartesian product of sets.
  - A relation can be considered as a "table" with rows and columns.

# Query Languages for the Relational Data Model

Codd introduced two different query languages for the relational data model:

- Relational Algebra, which is a procedural language.
  - It is an algebraic formalism in which queries are expressed by applying a sequence of operations to relations.
- Relational Calculus, which is a declarative language.
  - It is a logical formalism in which queries are expressed as formulas of first-order logic.

Codd's Theorem: Relational Algebra and Relational Calculus are *essentially equivalent in terms of expressive power*.

DBMSs are based on yet another language, namely SQL, a hybrid of a procedural and a declarative language that combines features from both relational algebra and relational calculus.

# The Five Basic Operations of Relational Algebra

Operators of Relational Algebra:

- Group I: Three standard set-theoretic binary operations:
  - Union
  - Difference
  - Cartesian Product
- Group II. Two special unary operations on relations:
  - Projection
  - Selection

*Note: Renaming can be expressed by Projection*

- Relational Algebra consists of all expressions obtained by combining these five basic operations in syntactically correct ways.

- If you want to try using Relational Algebra, go to https://users.cs.duke.edu/~junyang/radb/

# Relational Algebra: Standard Set-Theoretic Operations

- Union
  - Input: Two k-ary relations R and S, for some k.
  - Output: The k-ary relation $R \cup S$, where
    $R \cup S = \{(a_1,\ldots,a_k): (a_1,\ldots,a_k) \text{ is in } R \text{ or } (a_1,\ldots,a_k) \text{ is in } S\}$
- Difference:
  - Input: Two k-ary relations R and S, for some k.
  - Output: The k-ary relation $R - S$, where
    $R - S = \{(a_1,\ldots,a_k): (a_1,\ldots,a_k) \text{ is in } R \text{ and } (a_1,\ldots,a_k) \text{ is not in } S\}$

- Note:
  - In relational algebra, both arguments of the union and the difference must be relations of the same arity.
  - In SQL systems, there might be the additional requirement that the corresponding attributes must have the same data type.
  - However, the corresponding attributes need not have the same names; the corresponding attribute in the result can be renamed arbitrarily.

# Union

## Employee

| Code | Name | Age |
|---|---|---|
| 7274 | Rossi | 42 |
| 7432 | Neri | 54 |
| 9824 | Verdi | 45 |

## Director

| Code | Name | Age |
|---|---|---|
| 9297 | Neri | 33 |
| 7432 | Neri | 54 |
| 9824 | Verdi | 45 |

## Employee ∪ Director

| Code | Name | Age |
|---|---|---|
| 7274 | Rossi | 42 |
| 7432 | Neri | 54 |
| 9824 | Verdi | 45 |
| 9297 | Neri | 33 |

# Difference

**Employee**

| Code | Name | Age |
|---|---|---|
| 7274 | Rossi | 42 |
| 7432 | Neri | 54 |
| 9824 | Verdi | 45 |

**Director**

| Code | Name | Age |
|---|---|---|
| 9297 | Neri | 33 |
| 7432 | Neri | 54 |
| 9824 | Verdi | 45 |

**Employee – Director**

| Code | Name | Age |
|---|---|---|
| 7274 | Rossi | 42 |

# Relational Algebra: Cartesian Product

- Cartesian Product
  - Input: An m-ary relation R and an n-ary relation S
  - Output: The (m+n)-ary relation R × S, where

    $R \times S = \{(a_1,\ldots,a_m,b_1,\ldots,b_n)$: $(a_1,\ldots a_m)$ is in R and $(b_1,\ldots,b_n)$ is in S$\}$

- Note:

$$|R \times S| = |R| \times |S|$$

# Relational Algebra: Cartesian Product

**Employee**

| Emp | Dept |
|---|---|
| Rossi | A |
| Neri | B |
| Bianchi | B |

**Dept**

| Code | Chair |
|---|---|
| A | Mori |
| B | Bruni |

**Employee × Dept**

| Emp | Dept | Code | Chair |
|---|---|---|---|
| Rossi | A | A | Mori |
| Rossi | A | B | Bruni |
| Neri | B | A | Mori |
| Neri | B | B | Bruni |
| Bianchi | B | A | Mori |
| Bianchi | B | B | Bruni |

# The Projection Operation

- Motivation: It is often the case that, given a table R, one wants to rearrange the order of the columns and/or suppress/rename some columns
- Projection is a family of unary operations of the form

$$\pi_{\text{<attribute list>}}(\text{<relation name>})$$

or

$$\text{PROJ}_{\text{<attribute list>}}(\text{<relation name>})$$

- The intuitive description of the projection operation is as follows:
  - When the projection is applied to a relation R, it removes all columns whose attributes do not appear in the <attribute list>
  - The remaining columns may be re-arranged (and also renamed by means of the notation a ← b) according to the order and name in the <attribute list>
  - Any duplicate rows are eliminated

# The Projection Operation

– Show name and Site of employees

**Employee**

| Name | Site |
|---|---|
| Neri | Napoli |
| Neri | Milano |
| Rossi | Roma |

$$\text{PROJ}_{\text{Name, Site}}(\text{Employee})$$

# The Projection Operation

– Show name and Site of employees

**Employee**

| Name | Site |
|---|---|
| Neri | Napoli |
| Neri | Milano |
| Rossi | Roma |

**PROJ $_{\text{Name, Site}}$(Employee)**

– To rename: PROJ $_{N \leftarrow \text{Name}, A \leftarrow \text{Age}}$ (Employee)

# More on the Syntax of the Projection Operation

- In relational algebra, attributes can be referenced by position number
- Projection Operation:
  - Syntax: $\pi_{i_1,\ldots,i_m}(R)$, where R is of arity k, and $i_1,\ldots,i_m$ are distinct integers from 1 up to k.
  - Semantics:

$$\pi_{i_1,\ldots,i_m}(R) = \{ (a_1,\ldots,a_m): \text{there is a tuple } (b_1,\ldots,b_k) \text{ in R such that } a_1=b_{i_1}, \ldots, a_m=b_{i_m} \}$$

- Example: If R is R(A,B,C,D), then $\pi_{C,A}(R) = \pi_{3,1}(R)$

$\pi_{3,1}(R) = \{(a_1,a_2)$: there is (a,b,c,d) in R such that $a_1=c$ and $a_2=a\}$

# The Selection Operation

- Motivation: Given SAVINGS(branch-name, acc-no, cust-name, balance) we may want to extract the following information from it:
  - Find all records in the Aptos branch
  - Find all records with balance at least $50,000
  - Find all records in the Aptos branch with balance less than $1,000
- Selection is a family of unary operations of the form

  $\sigma_{\Theta}(R)$ OR $SEL_{\Theta}(R)$

  where R is a relation and $\Theta$ is a condition that can be applied as a test to each row of R.
- When a selection operation is applied to R, it returns the subset of R consisting of all rows that satisfy the condition $\Theta$
- Question: What is the precise definition of a “condition”?

# The Selection Operation

- Definition: A condition in the selection operation is an expression built up from:
  - Comparison operators =, <, >, ≠, ≤, ≥ applied to operands that are constants or attribute names or component numbers.
    - These are the basic (atomic) clauses of the conditions.
  - The Boolean logic operators ∧, ∨, : applied to basic clauses.
- Examples:
  - balance > 10,000
  - branch-name = “Aptos”
  - (branch-name = “Aptos”) ∧ (balance < 1,000)

# The Selection Operator

- Note:
  - The use of the comparison operators <, >, ≤, ≥ assumes that the underlying domain of values is totally ordered.
  - If the domain is not totally ordered, then only = and ≠ are allowed.
  - If we do not have attribute names (hence, we can only reference columns via their component number), then we need to have a special symbol, say $, in front of a component number. Thus,
    - $4 > 100 is a meaningful basic clause
    - $1 = “Aptos” is a meaningful basic clause, and so on.

# The Selection Operator

– Show the employees whose salary is greater than 50

## Employee

| Code | Name | Site | Salary |
|---|---|---|---|
| 7309 | Rossi | Roma | 55 |
| 5998 | Neri | Milano | 64 |
| 5698 | Neri | Napoli | 64 |

$\sigma_{Salary > 50}$ (Employee)

# Relational Algebra Expression

- Definition: A relational algebra expression is an expression obtained from relation schemas using union, difference, cartesian product, projection, and selection.
- Context-free grammar for relational algebra expressions:

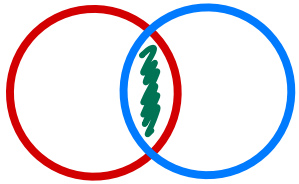
  $E := R, S, \ldots \mid (E_1 \cup E_2) \mid (E_1 - E_2) \mid (E_1 \times E_2) \mid \pi_X(E) \mid \sigma_\Theta(E)$,

  where
  - R, S, … are relation schemas
  - X is a list of attributes, possibly renamed
  - $\Theta$ is a condition.

# Derived Operation: Intersection

- Intersection
  - Input: Two k-ary relations R and S, for some k.
  - Output: The k-ary relation $R \cap S$, where

$$R \cap S = \{(a_1,\ldots,a_k): (a_1,\ldots,a_k) \text{ is in R and } (a_1,\ldots,a_k) \text{ is in S}\}$$



- Fact: $R \cap S = R - (R - S) = S - (S - R)$

Thus, intersection is a derived relational algebra operation.

# Intersection: example

**Employee**

| Code | Name | Age |
|---|---|---|
| 7274 | Rossi | 42 |
| 7432 | Neri | 54 |
| 9824 | Verdi | 45 |

**Director**

| Code | Name | Age |
|---|---|---|
| 9297 | Neri | 33 |
| 7432 | Neri | 54 |
| 9824 | Verdi | 45 |

**Employee ∩ Director**

| Code | Name | Age |
|---|---|---|
| 7432 | Neri | 54 |
| 9824 | Verdi | 45 |

# Derived Operation: Θ–Join and Beyond

Definition: A Θ-Join is a relational algebra expression of the form

$$\sigma_{\Theta}(R \times S) \quad \text{OR} \quad R \text{ JOIN}_{\Theta} S$$

Note:

- If R and S have attribute A in common, then we use the notation R.A and S.A to disambiguate.

- The Θ-Join selects those tuples from R × S that satisfy the condition Θ. In particular, if every tuple in R × S satisfies Θ, then

$$\sigma_{\Theta}(R \times S) = R \times S$$

# Θ–Join and Beyond

- Θ-joins are often combined with projection to express interesting queries.
- Example: F(name, dpt, salary), C(dpt, name), where F stands for FACULTY and C stands for CHAIR
  - Find department and salary of department chairs

    C-SALARY(dpt,salary) =

    $$\pi_{F.dpt,\ F.salary}(\sigma_{F.name = C.name}(F \times C))$$

Note: The Θ-Join in this example is an equijoin, since Θ is a conjunction of equality basic clauses.

Exercise: Show that the intersection R ∩ S can be expressed using a combination of projection and an equijoin.

# Θ–Join and Beyond

Example: F(name, dpt, salary), C-SALARY(dpt, salary)

Find the names of all faculty members of the EE department who earn a bigger salary than their department chair.

HIGHLY-PAID-IN-EE(Name) =

$\pi_{F.name}(\sigma_{F.dpt = \text{“EE”} \wedge F.dpt = C.dpt \wedge F.salary > C.salary}(F \times C\text{-SALARY}))$

Note: The Θ-Join above is not an equijoin.

# Derived Operation: Natural Join

The natural join between two relations is essentially the equi-join on common attributes (attributes of the two relations having the same name), projected in such a way that the attributes of the relation that is the result of the natural join is the union of attributes of the two relations.

Given T(facname, course, term), where T stands for TEACHES, and E(studname, course, term) where E stands fopr ENROLLS, we compute the natural join TAUGHT-BY(studname, course, term, facname) by:

$$\pi_{\text{E.studname, E.course, E.term, T.facname}} (\sigma_{\text{T.course = E.course} \wedge \text{T.term = E.term}} (\text{ENROLLS} \times \text{TEACHES}))$$

The resulting expression can be written using this notation:

ENROLLS $\bowtie$ TEACHES OR ENROLLS JOIN TEACHES

# Natural Join

- Definition: Let A1, …, Ak be the common attributes of two relation schemas R and S. Then

$$R \bowtie S = \pi_{<list>} (\sigma_{R.A1=S.A1 \wedge \ldots \wedge R.A1=S.Ak}(R \times S)),$$

where <list> contains the set of all attributes of $R \times S$, except for S.A1, …, S.Ak (in other words, duplicate columns are eliminated).

- Naive algorithm for $R \bowtie S$:

  For every tuple t in R, compare t with every tuple in S as follows:
  - test if they agree on all common attributes of R and S;
  - if they do, take the tuple in $R \times S$ formed by these two tuples,
    - remove all values of attributes of S that also occur in R;
    - put the resulting tuple in $R \bowtie S$.

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)

School(scode,city)

where School stores all the schools.

and write the Relational Algebra queries for:

1. Find the cities with at least one school with a student who graduated with 100.
2. Find the schools where no student has graduated with 100.
3. Find the cities where all schools have a student who graduated with 100
4. For all school, find the student(s) who graduated with the minimum grade in the school.

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the Relational Algebra query for:

1. Find the cities with at least one school with a student who graduated with 100.

$$\text{PROJ}_{\text{CITY}}\left(\sigma_{\text{MARK}=100 \wedge \text{SCHOOL}=\text{SCODE}}(\text{GRADUATED} \times \text{SCHOOL})\right)$$

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the Relational Algebra query for:

1. Find the cities with at least one school with a student who graduated with 100.

$PROJ_{city}$ ($SEL_{mark=100}$ (Graduated) $JOIN_{school=scode}$ School))

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the Relational Algebra query for:

2. Find the schools where no student has graduated with 100.

$$PROJ_{SCODE}(SCHOOL) - PROJ_{SCHOOL}(\sigma_{MARK=100}(GRADUATED))$$

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the Relational Algebra query for:

2. Find the schools where no student has graduated with 100.

$PROJ_{scode}$ (School) – $PROJ_{school}$ ($SEL_{mark=100}$ (Graduated))

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the Relational Algebra query for:

2. Find the schools where no student has graduated with 100.

$PROJ_{scode}$ (School) – $PROJ_{school}$ ($SEL_{mark=100}$ (Graduated))

The scode of the schools with at least one student graduated with 100

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)

School(scode,city)

and write the Relational Algebra query for:

3. Find the cities where all schools have a student who graduated with 100

$$\pi_{city}(SCHOOL) - \pi_{city}(SCHOOL \bowtie (\pi_{scode}(SCHOOL) - \pi_{school}(\sigma_{mark=100}(GRADUATED))))$$

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)

School(scode,city)

and write the Relational Algebra query for:

3. Find the cities where all schools have a student who graduated with 100

The cities with some school with no student graduated with 100

$PROJ_{city}$ (School) –

$PROJ_{city}$

(School JOIN

The schools with no student graduated with 100

$PROJ_{scode}$ (School) – $PROJ_{school}$ ($SEL_{mark=100}$ (Graduated))

)

The schools with at least one student graduated with 100

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the Relational Algebra query for:

4. For all school, find the student(s) who graduated with the minimum grade in the school.

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the Relational Algebra query for:

4. For all school, find the student(s) who graduated with the minimum grade in the school.

$$PROJ_{school,gcode}\ (Graduated - PROJ_{gcode,mark,school}(Graduated\ JOIN_{mark>m\ and\ school=s} PROJ_{m \leftarrow mark,\ s \leftarrow school}\ (Graduated)))$$

# SQL: Structured Query Language

- SQL is the standard language for relational DBMSs
- We will present the syntax of the core SQL constructs and then will give rigorous semantics by interpreting SQL to Relational Algebra.
- Note: SQL typically uses multiset semantics, but we ignore this property here, and we only consider the set-based semantics (adopted by using the keyword DISTINCT in queries)

# SQL: Structured Query Language

- The basic SQL construct is:
  SELECT DISTINCT <attribute list>
  FROM <relation list>
  WHERE <condition>
- More formally,
  SELECT DISTINCT $R_{i1}$.A1, … , $R_{im}$.Am
  FROM $R_1$, … ,$R_K$
  WHERE $\gamma$

**Restrictions:**

- $R_1$, … ,$R_K$ are relation names (possibly, with aliases for renaming, where an alias N for relation name $R_i$ is denoted by $R_i$ AS N)
- Each $R_{ij}$.Aj is an attribute of $R_{ij}$
- $\gamma$ is a condition with a precise (and rather complex) syntax.

# SQL vs. Relational Algebra

| SQL | Relational Algebra |
|---|---|
| SELECT | Projection |
| FROM | Cartesian Product |
| WHERE | Selection |

Semantics of SQL via interpretation to Relational Algebra:

SELECT DISTINCT $R_{i1}.A1, \ldots, R_{im}.Am$

FROM $R_1, \ldots, R_K$

WHERE $\gamma$

corresponds to

$$\pi_{R_{i1}.A1, \ldots, R_{im}.Am} (\sigma_{\gamma} (R_1 \times \ldots \times R_K))$$

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)

School(scode,city)

where School stores all the schools.

and write the SQL queries for:

1. Find the cities with at least one school with a student who graduated with 100.
2. Find the schools where no student has graduated with 100.
3. Find the cities where all schools have a student who graduated with 100
4. For all school, find the student(s) who graduated with the minimum grade in the school.

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the SQL query for:

1. Find the cities with at least one school with a student who graduated with 100.

```
SELECT DISTINCT CITY
FROM   GRADUATED G, SCHOOL S
WHERE  G.MARK=100 AND G.SCHOOL=S.SCODE
```

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the SQL query for:

1. Find the cities with at least one school with a student who graduated with 100.

```
select city
from Graduated join School on school = scode
where mark = 100
```

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the SQL query for:

2. Find the schools where no student has graduated with 100.

```
SELECT SCODE
FROM SCHOOL
WHERE SCODE NOT IN (SELECT SCHOOL
                    FROM GRADUATED G
                    WHERE G.MARK=100)
```

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the SQL query for:

2. Find the schools where no student has graduated with 100.

```
select scode
from School
where scode not in (select school
                    from Graduate
                    where mark = 100)
```

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the SQL query for:

3. Find the cities where all schools have a student who graduated with 100

```
SELECT CITY FROM SCHOOL
EXCEPT
SELECT S.CITY
FROM SCHOOL S JOIN GRADUATED G ON S.SCODE = G.SCHOOL
WHERE S.SCODE NOT IN (SELECT SCHOOL
                      FROM GRADUATED G
                      WHERE G.MARK = 100)
```

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)
School(scode,city)

and write the SQL query for:

3. Find the cities where all schools have a student who graduated with 100

```
select city from School
where city not in (select city
                   from School
                   where scode not in (select scode from Graduate
                                       where mark = 100))
```

# Exercises

Consider a database with relations

Graduated(gcode,mark,school)

School(scode,city)

and write the SQL query for:

4. For all school, find the student(s) who graduated with the minimum grade in the school.

```
select g1.school, g1.gcode
from Graduated g1
where g1.mark =   (select min(mark)
                   from Graduated g2
                   where g1.school = g2.school)
```

# Aggregation operators

Among the expressions in the target list, we can have also expressions that compute values over a set of tuples:

**counting, minimum, maximum, average, sum**

*Function* **(** [ **DISTINCT** ] *ExpressionOverAttributes* **)**

# Aggregation operators: COUNT

- Count the number of tuples:

  **COUNT (*)**

- Count the values different from NULL of an attribute (considering duplicates):

  **COUNT (***Attribute***)**

- Count the distinct values different from NULL of an attribute:

  **COUNT (DISTINCT** *Attribute***)**

# Example

**paternity**

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

# Aggregation operator COUNT: Example and Semantics

*Example*: How many children does Franco have?

```
SELECT COUNT(*) AS numChildrenOfFranco
FROM   paternity
WHERE  father = 'Franco'
```

The aggregation operator (**COUNT**), which counts the tuples, is applied to the result of the following query:

```
SELECT *
FROM   paternity
WHERE  father = 'Franco'
```

# Result of COUNT: Example

paternity

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

| numChildrenOfFranco |
|---|
| 2 |

# COUNT and null values

```
SELECT COUNT(*)
FROM   persons
```

Result = number of **tuples**
= 4

```
SELECT COUNT(income)
FROM   persons
```

Result = number of **values different from NULL**
= 3

```
SELECT COUNT(DISTINCT income)
FROM   persons
```

Result = number of **distinct values different from NULL**
= 2

**persons**

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

# Other aggregation operators

**SUM, AVG, MAX, MIN**

- they admit as argument an attribute or an expression (but not "*****")
- **SUM** and **AVG**: numeric arguments or time
- **MAX** and **MIN**: arguments on which an order is defined

*Example*: average of the incomes of the children of Franco.

```
SELECT AVG(income)
FROM   persons JOIN paternity ON
         name = child
WHERE  father = 'Franco'
```

# Aggregation operators and null values

```
SELECT AVG(income) AS averageIncome
FROM   persons
```

**persons**

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

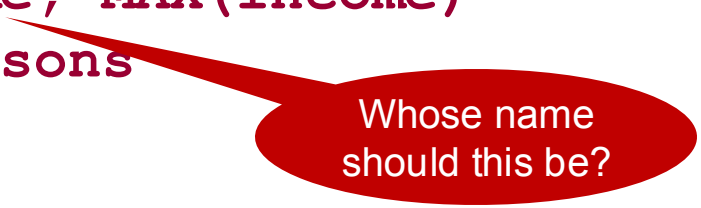
Is ignored

| averageIncome |
|---|
| 34 |

# Aggregation operators and target list

A query that **does not make sense**:

```
SELECT name, MAX(income)
FROM   persons
```




For the query to make sense, the **target list** must be **homogeneous**.

*Example:*

```
SELECT MIN(age), AVG(income)
FROM   persons
```

# Aggregation operators and grouping

- In the previous cases, the aggregation operators are applied to the set of all tuples that form the result.
- In many cases, we would like that the aggregations are applied to a **partition of the tuples** of the relations.
- To specify the partitions of the tuples, we make use of the `GROUP BY` clause:

**`GROUP BY`** *AttributeList*

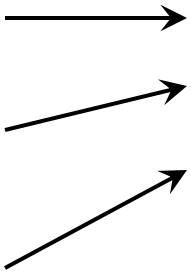
# Aggregation operators and grouping: Example

The number of children of each father.

```
SELECT father, COUNT(*) AS numChildren
FROM   paternity
GROUP BY father
```

**paternity**

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



| father | numChildren |
|---|---|
| Sergio | 1 |
| Luigi | 2 |
| Franco | 2 |

# Semantics of queries with aggregation operators and grouping

1. The query is executed **ignoring `GROUP BY`** and the aggregation operators:

```
SELECT *
FROM   paternity
```

2. The tuples that have **the same value for the attributes in the `GROUP BY` clause** are grouped together:
   - the result contains one tuple for each group, and
   - the aggregation operators are applied to each group.

# Exercise 7: GROUP BY

Maximum income for each group of persons of that are at least 18 years old and have the same age, indicating also the age.

Express the query in SQL.

**persons**

| name | age | income |
|---|---|---|

# Esercise 7: Solution

Maximum income for each group of persons of that are at least 18 years old and have the same age, indicating also the age.

```
SELECT age, MAX(income)
FROM   persons
WHERE  age >= 18
GROUP BY age
```

# Conditions on the groups

We can also impose **selection conditions on the groups**. A selection on the groups is clearly **different from** a condition that selects the tuples that have to form the groupts (**`WHERE`** clause). To carry out a selection on the groups, we use the **`HAVING`** clause, which has to appear after the **`GROUP BY`** clause.

*Example*: The fathers whose children have an average income greater than 25K.

```
SELECT father, AVG(income)
FROM   persons JOIN paternity ON
         child = name
GROUP BY father
HAVING AVG(income) > 25
```

# Exercise 8: WHERE or HAVING?

The fathers whose children younger than 30 have an average income greater than 20K.

# Exercise 8: Solution

The fathers whose children younger than 30 have an averge income greater than 20K.

```
SELECT father, AVG(income)
FROM   persons JOIN paternity ON
          child = name
WHERE  age < 30
GROUP BY father
HAVING AVG(income) > 20
```

# Syntax of `SELECT`: Summary

*SelectSQL* ::=

```
SELECT      AttributesOrExpressionsList
FROM        TablesList
[ WHERE     SimpleConditions ]
[ GROUP BY  GroupingAttributesList ]
[ HAVING    AggregationConditions ]
[ ORDER BY  SortAttributesList ]
[ LIMIT     number ]
```

# Grouping and target list

We have already observed that in a query that uses **`GROUP BY`**, the target list should be **homogeneous**, i.e., the target list should contain (in addition to the aggregation functions) **only** attributes that appear in the **`GROUP BY`** clause.

*Example*:

- Incomes of persons, grouped by age (**not reasonable**, since the target list is not homogeneous: there could be multiple values of income for the same group):

```
SELECT age, income
FROM persons
GROUP BY age
```

- Average of the incomes of persons, grouped by age (**reasonable**, since for each group there is only one average of the incomes):

```
SELECT age, AVG(income)
FROM persons
GROUP BY age
```

# Non-homogeneous target list

What happens if in a query using **`GROUP BY`**, the target list is not homogeneous?

Postgres signals an error, but some systems do not signal an error, and return one of the values that are associated to the current values of the attributes that form the group.

*Example:*

Income of persons, grouped according to age (**non-homegenous target list**, since there could be multiple values of income for the same group):

```
SELECT age, income
FROM persons
GROUP BY age
```

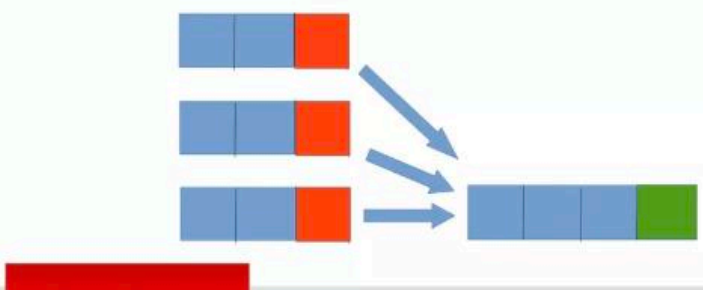
MySQL, for example, does not signal an error. It chooses for each group one if the values of income that appears in the group, and returns it for the attribute **`income`** in the target list.

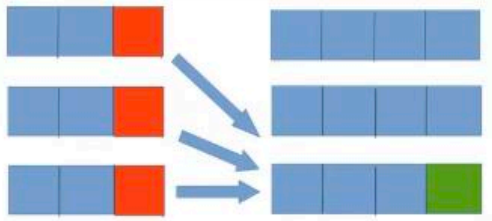
# The notion of window

- Window functions perform calculations across a set of table rows related to the current row.
- Unlike aggregate functions, they do not collapse the result into a single row.
- They use the OVER() clause to define the window of rows for computation.
- Common window functions: ROW_NUMBER(), RANK(), DENSE_RANK(), SUM(), AVG(), LEAD(), LAG().

# Window functions: what are they?

- A window function performs a calculation across a set of rows that are related to the current row, similar to what can be done with an aggregate function.
- But unlike traditional aggregate functions, a window function does not cause rows to become grouped into a single output row.
- So, similar to normal function, but can access values of other rows “in the vicinity” of the current row





# Window function example

```
SELECT name, department_id, salary,
       SUM(salary) OVER (PARTITION BY department_id)
                        AS department_total
       FROM employee
       ORDER BY department_id, name;
```

The OVER keyword signals a window function

## Partitions: 10, 20, 30

| name | department_id | salary | department_total |
|---|---|---|---|
| Newt | NULL | 75000 | 75000 |
| Dag | 10 | NULL | 370000 |
| Ed | 10 | 100000 | 370000 |
| Fred | 10 | 60000 | 370000 |
| Jon | 10 | 60000 | 370000 |
| Michael | 10 | 70000 | 370000 |
| Newt | 10 | 80000 | 370000 |
| Lebedev | 20 | 65000 | 130000 |
| Pete | 20 | 65000 | 130000 |
| Jeff | 30 | 300000 | 370000 |
| Will | 30 | 70000 | 370000 |

Partition == disjoint set of rows in result set

Here: all rows in partition are peers

# With GROUP BY

```
SELECT name, department_id, salary,
       SUM(salary) AS department_total
       FROM employee GROUP BY department_id
       ORDER BY department_id, name;
```

```
ERROR 1055 (42000): Expression #1 of SELECT list is not in GROUP BY clause and
contains nonaggregated column 'mysql.employee.name' which is not functionally dependent
on columns in GROUP BY clause; this is incompatible with sql_mode=only_full_group_by
```

# With GROUP BY

```
SELECT /* name, */ department_id, /* salary,*/
       SUM(salary) AS department_total
       FROM employee GROUP BY department_id
       ORDER BY department_id /*, name */;
```

```
+---------------+------------------+
| department_id | department_total |
+---------------+------------------+
|          NULL |            75000 |
|            10 |           370000 |
|            20 |           130000 |
|            30 |           370000 |
+---------------+------------------+
```

# Components of Window Functions

- Function (e.g., SUM, AVG, ROW_NUMBER)
- OVER() clause defining:
  - PARTITION BY - divides rows into groups
  - ORDER BY - specifies order within partitions
  - ROWS/RANGE - optional frame specification

# Exercise: can we express this query without the notion of window?

## Window function example

```
SELECT name, department_id, salary,
       SUM(salary) OVER (PARTITION BY department_id)
                        AS department_total
       FROM employee
       ORDER BY department_id, name;
```

The OVER keyword signals a window function

# Example: ROW_NUMBER()

Assigns a unique sequential integer to rows within a partition.

Example:

```
SELECT employee_id, department, salary,
      ROW_NUMBER() OVER
      (PARTITION BY department ORDER BY salary
DESC) AS row_num
FROM employees;
```

Usage: Useful for pagination and ranking employees within departments.

# Example: RANK() vs DENSE_RANK()

RANK() assigns a rank but skips numbers for ties.
DENSE_RANK() does not.

Example:

```
SELECT employee_id, department, salary,
      RANK() OVER
            (PARTITION BY department ORDER BY salary
DESC) AS rank,
      DENSE_RANK() OVER
            (PARTITION BY department ORDER BY salary
DESC) AS dense_rank
FROM employees;
```

Usage: Helps in ranking employees with tied salaries.

# Visualization: ROW_NUMBER(), RANK(), DENSE_RANK()

Example Data:

| Employee | Department | Salary | ROW_NUMBER() | RANK() | DENSE_RANK() |
|---|---|---|---|---|---|
| A | HR | 5000 | 1 | 1 | 1 |
| B | HR | 5000 | 2 | 1 | 1 |
| C | HR | 4500 | 3 | 3 | 2 |
| D | HR | 4000 | 4 | 4 | 3 |

RANK() skips 2 after the tie, DENSE_RANK() does not.

# Example: SUM() with OVER()

Computes cumulative or partitioned aggregates without collapsing rows.

Example:

```
SELECT employee_id, department, salary,
       SUM(salary) OVER (PARTITION BY department)
       AS total_salary
FROM employees;
```

Usage: Computes total salary per department without aggregation.

# Example: LEAD() and LAG()

LEAD() fetches the next row's value; LAG() fetches the previous row's value.

Example:

```
SELECT employee_id, department, salary,
      LAG(salary) OVER
         (PARTITION BY department ORDER BY salary) AS prev_salary,
      LEAD(salary) OVER
         (PARTITION BY department ORDER BY salary) AS next_salary
FROM employees;
```

Usage: Compare salaries between consecutive employees.

# Practical Use Cases

- Pagination with ROW_NUMBER() for web applications.
- Ranking employees/customers based on sales using RANK().
- Finding previous/next purchase amount using LAG()/LEAD().
- Calculating running totals with SUM() OVER().
- Identifying percentile rankings for salaries.

# Comments

- Window functions provide powerful analytics while retaining row granularity.
- The OVER() clause controls how data is grouped and ordered.
- Functions like ROW_NUMBER, RANK, SUM, LEAD, and LAG enhance SQL queries.
- Useful for advanced reporting, BI, and data analysis tasks.