

# Chapter 3

# Design theory for Relational Databases

Université Grenoble Alpes

22/03/2023

Bahareh Afshinpour

[bahareh.afshinpour@univ-grenoble-alpes.fr](mailto:bahareh.afshinpour@univ-grenoble-alpes.fr)

Main reference:

*A First Course in Database Systems* (and associated material) by  
J. Ullman and J. Widom, Prentice-Hall

# Examples

- **lives**(person-name,street,city)
- **works**(person-name, company-name,salary)
- **located-in**(company-name,city)
- **manages**(person-name,manager-name)

For the above schema (the primary key for each relation is denoted by the underlined attribute), provide relational algebra expressions for the following queries:

1. Find all tuples in works of all persons who work for the City Bank company (which is a specific company in the database).

(a)  $\sigma_{(cname='City Bank')}(works)$

2. Find the name of persons working at City Bank who earn more than \$50,000.

(a)  $\pi_{pname}(\sigma_{(cname='City Bank') \wedge (salary > 50000)}(works))$

3. Find the name and city of all persons who work for City Bank and earn more than 50,000. Similar to previous query, except we have to access the lives table to extract the city of the employee . Note the join condition in the query.

(a)  $\pi_{lives.pname,lives.city}(\sigma_{((cname='City Bank') \wedge (salary > 50000) \wedge (lives.pname=works.pname))}(lives \times works))$

Find names of all persons who do not work for City Bank. Can write this in multiple ways  
- one solution is to use set difference:

$$(a) (\pi_{pname}(works)) - (\pi_{pname}(\sigma_{cname='City Bank'}(works)))$$

6. Find the name of all persons who work for City Bank and live in DC. Similar to query 3, but select only with tuples where person city is DC.

$$(a) \pi_{lives.pname}(\sigma_{((cname='City Bank') \wedge (lives.city='DC') \wedge (lives.pname=works.pname))})(lives \times works)$$

# SQL – Structured Query Language

- The most commonly used relational DBMS'S query and modify the database through a language called SQL.
- The portion of SQL that supports queries has capabilities very close to relational algebra.

# SQL – Structured Query Language

- Perhaps the simplest form of query in SQL asks for those tuples of some one relation that satisfy a condition.
- This simple query, like almost all SQL queries, uses the three keywords. **SELECT**, **FROM**, and **WHERE** that characterize SQL.

```
SELECT *  
FROM Movies  
WHERE studioName = 'Disney' AND year = 1990;
```

## A Trick for Reading and Writing Queries

It is generally easiest to examine a select-from-where query by first looking at the **FROM** clause, to learn which relations are involved in the query. Then, move to the **WHERE** clause, to learn what it is about tuples that is important to the query. Finally, look at the **SELECT** clause to see what the output is. The same order — from, then where, then select — is often useful when writing queries of your own, as well.

# Projection in SQL

**Example 6.2:** Suppose we wish to modify the query of Example 6.1 to produce only the movie title and length. We may write

```
SELECT title, length
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

The result is a table with two columns, headed `title` and `length`. The tuples in this table are pairs, each consisting of a movie title and its length, such that the movie was produced by Disney in 1990. For instance, the relation schema and one of its tuples looks like:

<i>title</i>	<i>length</i>
Pretty Woman	119
...	...

---

<sup>1</sup>Thus, the keyword **SELECT** in SQL actually corresponds most closely to the projection operator of relational algebra, while the selection operator of the algebra corresponds to the **WHERE** clause of SQL queries.

# Projection in SQL

- Sometimes, we wish to produce a relation with **column headers different** from the attributes of the relation mentioned in the From clause.
- We may follow the name of the attribute by the keyword **AS** and an alias, which becomes the header in the result relation.

**Example 6.3:** We can modify Example 6.2 to produce a relation with attributes `name` and `duration` in place of `title` and `length` as follows.

```
SELECT title AS name, length AS duration
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

The result is the same set of tuples as in Example 6.2, but with the columns headed by attributes `name` and `duration`. For example,

<i>name</i>	<i>duration</i>
Pretty Woman	119
...	...

# Projection in SQL

- We can use an expression in place of an attribute.

**Example 6.4:** Suppose we want output as in Example 6.3, but with the length in hours. We might replace the SELECT clause of that example with

```
SELECT title AS name, length*0.016667 AS lengthInHours
```

Then the same movies would be produced, but lengths would be calculated in hours and the second column would be headed by attribute `lengthInHours`, as:

<i>name</i>	<i>lengthInHours</i>
Pretty Woman	1.98334
...	...

- Lengths would be calculated in hours
- Then rename



# Case Insensitivity

## Case Insensitivity

SQL is *case insensitive*, meaning that it treats upper- and lower-case letters as the same letter. For example, although we have chosen to write keywords like **FROM** in capitals, it is equally proper to write this keyword as **From** or **from**, or even **FrOm**. Names of attributes, relations, aliases, and so on are similarly case insensitive. **Only inside quotes does SQL** make a distinction between upper- and lower-case letters. Thus, **'FROM'** and **'from'** are different character strings. Of course, neither is the keyword **FROM**.

# Selection

- WHERE clause `<attribute> <operator> <value>`
- We may build expressions by comparing values using the six common comparison operators: `=`, `<>`, `>`, `<`, `<=`, `>=`

Not equal

```
vol.depart = "Londres"  
avion.cap < '300'  
avion.type = 'AIRBUS 300'
```

# Selection

## SQL Queries and Relational Algebra

The simple SQL queries that we have seen so far all have the form:

```
SELECT L
FROM R
WHERE C
```

in which  $L$  is a list of expressions,  $R$  is a relation, and  $C$  is a condition. The meaning of any such expression is the same as that of the relational-algebra expression

$$\pi_L(\sigma_C(R))$$

That is, we start with the relation in the **FROM** clause, apply to each tuple whatever condition is indicated in the **WHERE** clause, and then project onto the list of attributes and/or expressions in the **SELECT** clause.

# Selection Example

```
Select pilote.nom  
From pilote  
Where pilote.prenom = 'Antoine';
```

```
Select pilote.nom  
From pilote  
Where pilote.prenom = 'Antoine';
```

```
Select pilote.nom  
From pilote  
Where pilote.prenom = 'Antoine';
```

PILOTE

numpilote	nom	prenom
P0001	Dupuis	Antoine
P0002	Simon	Georges
P0003	François	Luc
P0004	André	Georges
P0005	Arthur	Louis
P0006	Mathieu	François

numpilote	nom	prenom
P0001	Dupuis	Antoine
P0002	Simon	Georges
P0003	François	Luc
P0004	André	Georges
P0005	Arthur	Louis
P0006	Mathieu	François

numpilote	nom	prenom
	Dupuis	

# SELECT Statement

Used for queries on single or multiple tables

Clauses of the SELECT statement:

## +SELECT

× List the columns (and expressions) to be returned from the query

## +FROM

× Indicate the table(s) or view(s) from which data will be obtained

## +WHERE

× Indicate the conditions under which a row will be included in the result

## +GROUP BY

× Indicate categorization of results

## +HAVING

× Indicate the conditions under which a category (group) will be included

## +ORDER BY

× Sorts the result according to specified criteria

# Aggregation operators (summarize)

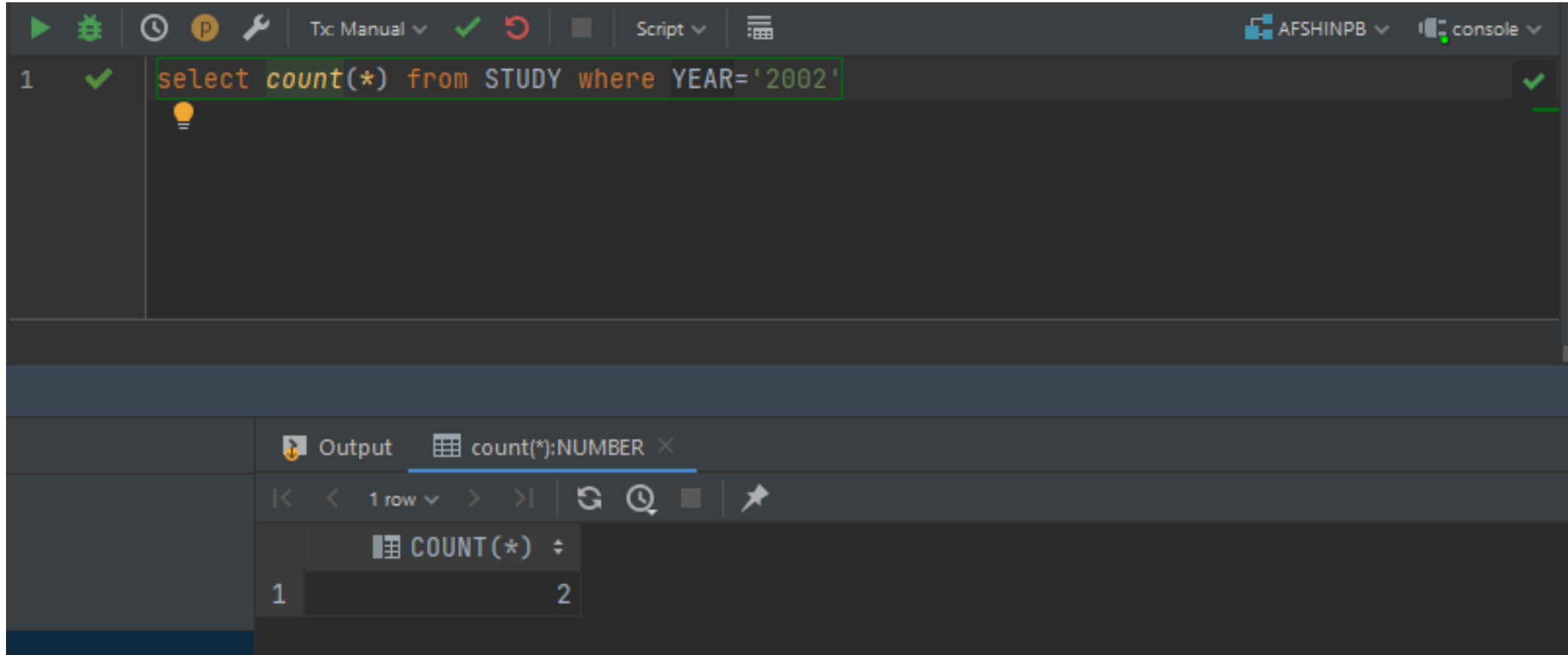
- Many operators we can apply to set or bags of numbers or strings.
- They are used to **summarize or aggregate** the values in one column of relation
  
- For example:
- SUM
- AVG
- MIN and MAX (numerical values and character-string values)
- COUNT

To retrieve the number of person and their salary

**Count(FName) R    AVERAGE(Salary) R**

FName	SSN	Salary	DNo
Ann	123654789	40000	2
Jeremy	969687423	30000	2
Peter	333265874	30000	1
Elsa	888548623	20000	2

# TP1- How many papers (studies) do we have in 2002?



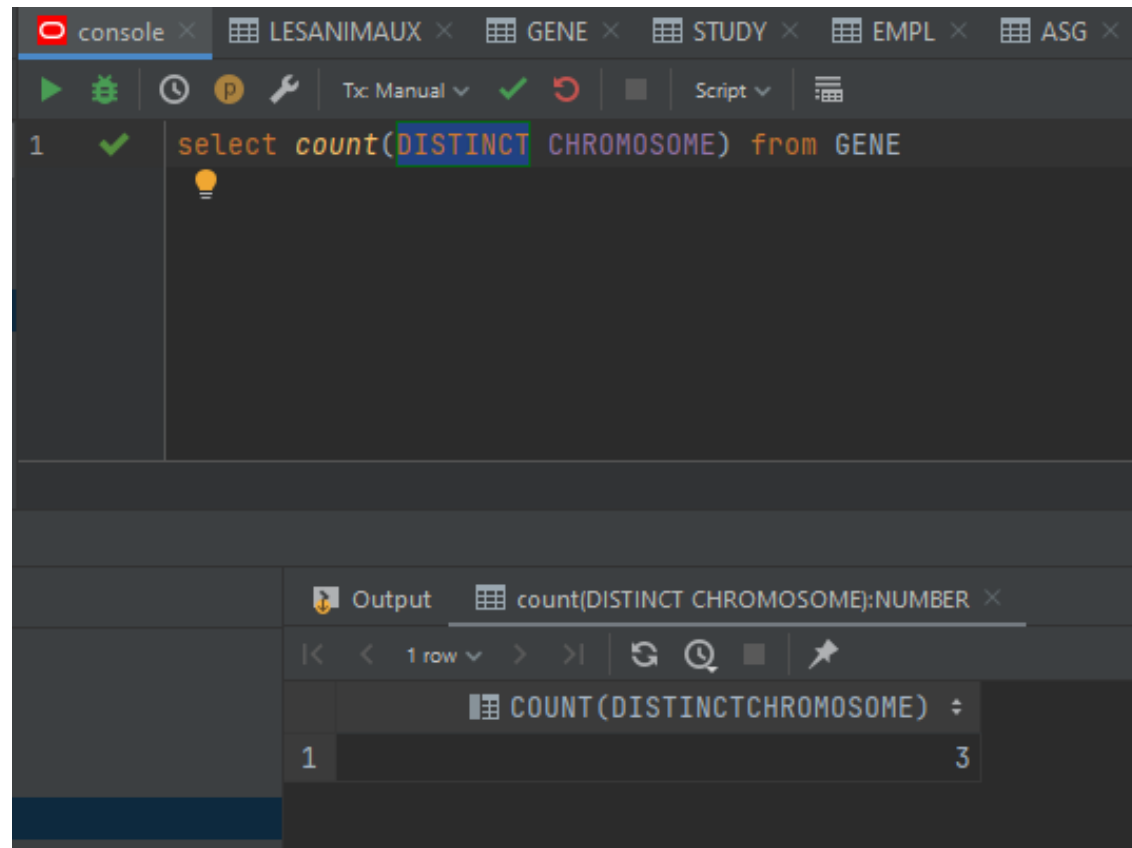
The screenshot shows a SQL IDE interface. The top toolbar includes icons for execution, debugging, and manual refresh. The main editor area contains a single SQL query: `select count(*) from STUDY where YEAR='2002'`. Below the editor, the 'Output' panel is active, displaying the result of the query in a table format. The table has one row with the value 2.

```
select count(*) from STUDY where YEAR='2002'
```

COUNT(*)
2

# Eliminating Duplicates in an Aggregation

- Use **DISTINCT** inside an aggregation.
- Example: find the number of the different chromosomes that we have genes in GENE table:



The screenshot shows a SQL IDE interface with a console window. The console displays a SQL query: `select count(DISTINCT CHROMOSOME) from GENE`. The word `DISTINCT` is highlighted in blue. Below the query, the output is shown in a table with one row and two columns. The first column is labeled `COUNT(DISTINCTCHROMOSOME)` and the second column contains the value `3`.

```
1 ✓ select count(DISTINCT CHROMOSOME) from GENE
```

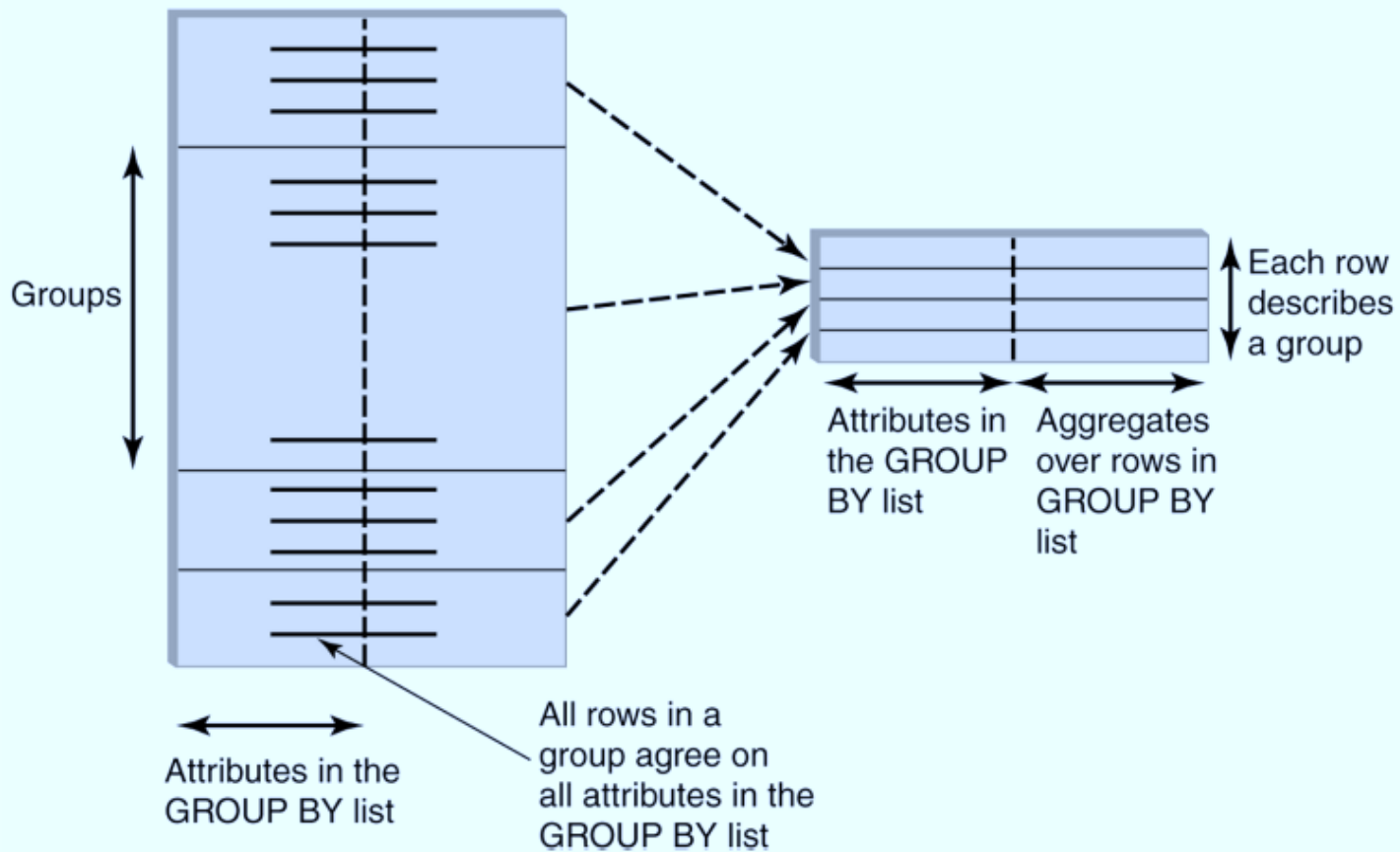
	COUNT(DISTINCTCHROMOSOME)
1	3



# The Grouping Operator $\gamma_L(R)$

The relation returned by the expression  $\gamma_L(R)$  is constructed as follows:

1. Partition the tuples of  $R$  into *groups*. Each group consists of all tuples having one particular assignment of values to the grouping attributes in the list  $L$ . If there are no grouping attributes, the entire relation  $R$  is one group.
2. For each group, produce one tuple consisting of:
  - i.* The grouping attributes' values for that group and
  - ii.* The aggregations, over all tuples of that group, for the aggregated attributes on list  $L$ .



### Transcript

1234		1234	3.3
1234			
1234			
1234			

- Attributes:*
- student's *Id*
  - avg grade
  - number of courses

# GROUP BY

The **GROUP BY** statement groups rows that have the same values into summary rows

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s);
```

	NOMA	SEXE	TYPE	PAYS	ANNAIS	NOCAGE
1	Charly	male	lion	Kenya	1999	12
2	Arthur	male	ours	France	2000	1
3	Chloe	femelle	pie	France	2001	3
4	Milou	male	leopard	France	2013	11
5	Tintin	male	leopard	France	2013	11
6	Charlotte	femelle	lion	Kenya	2012	12
7	Huan	femelle	panda	Chine	2005	1
8	Yuan Meng	male	panda	France	2015	1
9	Lola	femelle	lion	Kenya	1999	5
10	Tola	femelle	ours	Espagne	2003	1
11	Tai Lung	femelle	leopard	Chine	2006	5

The screenshot shows a SQL IDE interface with a console window. The console displays the following SQL query:

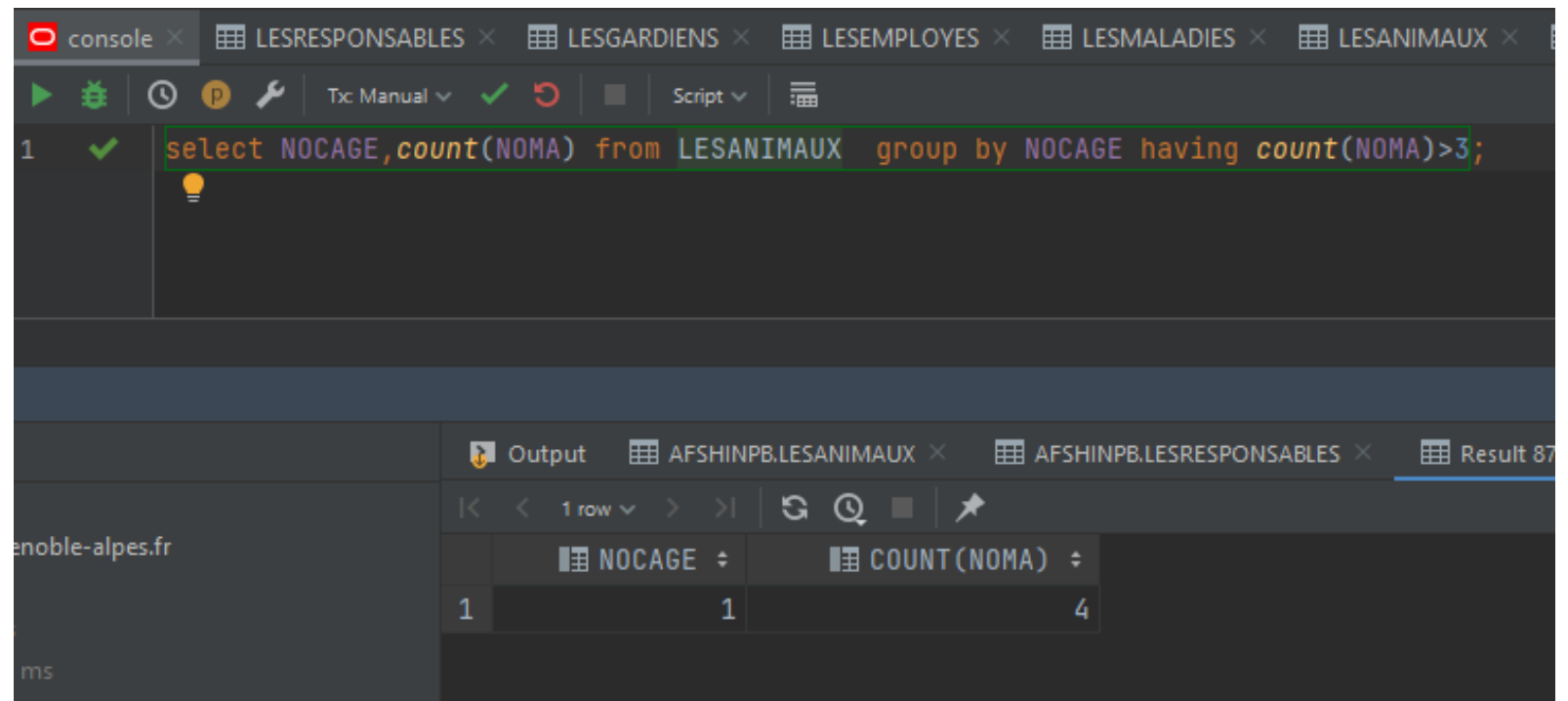
```
select NOCAGE, count(NOMA) from LESANIMAUX group by NOCAGE
```

Below the query, the results are displayed in a table format:

	NOCAGE	COUNT(NOMA)
1	11	2
2	12	2
3	3	1
4	5	2
5	1	4

The **HAVING** clause was added to SQL because the **WHERE** keyword cannot be used with aggregate functions.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition;
```



The screenshot shows a SQL IDE interface with a console window. The console displays a SQL query: `select NOCAGE, count(NOMA) from LESANIMAUX group by NOCAGE having count(NOMA)>3;`. Below the query, the result is shown as a table with two columns: `NOCAGE` and `COUNT(NOMA)`. The table contains one row with the values 1 and 4.

	NOCAGE	COUNT(NOMA)
1	1	4

# Extending the projection Operator

An expression  $x \rightarrow y$ , where  $x$  and  $y$  are names for attributes. The element  $x \rightarrow y$  in the list  $L$  asks that we take the attribute  $x$  of  $R$  and **rename** it  $y$ ; i.e., the name of this attribute in the schema of the result relation is  $y$ .

An expression  $E \rightarrow z$ , where  $E$  is an expression involving attributes of  $R$ , constants, arithmetic operators, and string operators, and  $z$  is a name for the attribute that results from the calculation implied by  $E$ . Example,  $a + b \rightarrow x$  as a list element represents the sum of the attributes  $a$  and  $b$ , renamed  $x$ . Element  $c || d \rightarrow e$  means concatenate the string-valued attributes  $c$  and  $d$  and call the result  $e$ .

**Example 5.11:** Let  $R$  be the relation

$A$	$B$	$C$
0	1	2
0	1	2
3	4	5

Then the result of  $\pi_{A, B+C \rightarrow X}(R)$  is

$A$	$X$
0	3
0	3
3	9

# The sorting Operator(Ordering the output)

- More efficient
- More easily find tuple

The expression  $\tau_L(R)$ , where  $R$  is a relation and  $L$  a list of some of  $R$ 's attributes, is the relation  $R$ , but with the tuples of  $R$  sorted in the order indicated by  $L$ . If  $L$  is the list  $A_1, A_2, \dots, A_n$ , then the tuples of  $R$  are sorted first by their value of attribute  $A_1$ . Ties are broken according to the value of  $A_2$ ; tuples that agree on both  $A_1$  and  $A_2$  are ordered according to their value of  $A_3$ , and so on. Ties that remain after attribute  $A_n$  is considered may be ordered arbitrarily.

**Example 5.12:** If  $R$  is a relation with schema  $R(A, B, C)$ , then  $\tau_{C,B}(R)$  orders the tuples of  $R$  by their value of  $C$ , and tuples with the same  $C$ -value are ordered by their  $B$  value. Tuples that agree on both  $B$  and  $C$  may be ordered arbitrarily. □

**Example 6.11:** The following is a rewrite of our original query of Example 6.1, asking for the Disney movies of 1990 from the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

To get the movies listed by length, shortest first, and among movies of equal length, alphabetically, we can say:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

**Exercise 6.1.3:** Write the following queries in SQL. They refer to the database schema of Exercise 2.4.1:

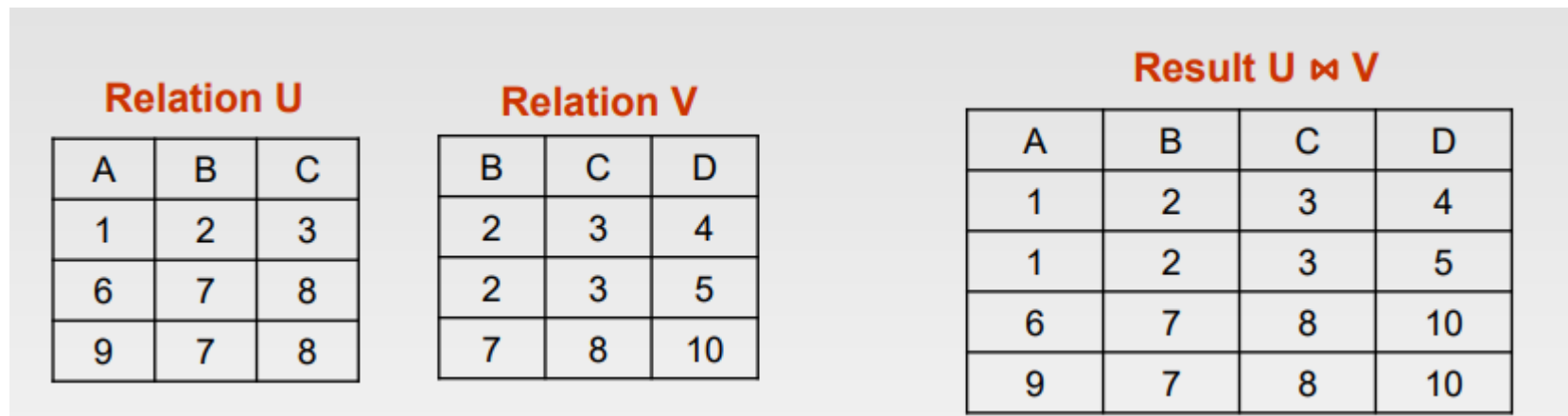
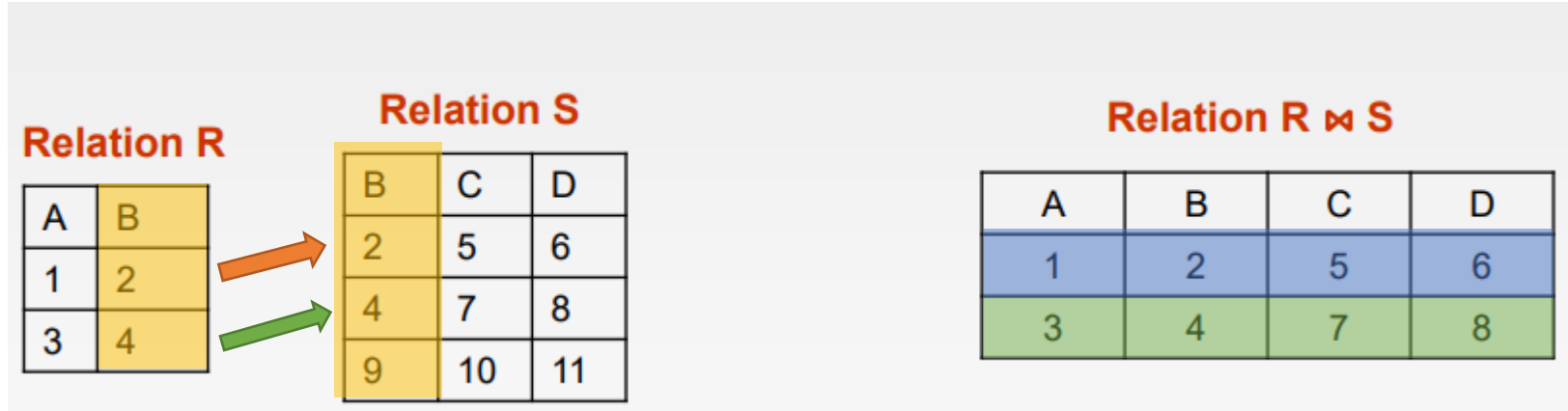
```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

Show the result of your queries using the data from Exercise 2.4.1.

- a) Find the model number, speed, and hard-disk size for all PC's whose price is under \$1000.
- b) Do the same as (a), but rename the **speed** column **gigahertz** and the **hd** column **gigabytes**.
- c) Find the manufacturers of printers.
- d) Find the model number, memory size, and screen size for laptops costing more than \$1500.
- e) Find all the tuples in the **Printer** relation for color printers. Remember that **color** is a boolean-valued attribute.
- f) Find the model number and hard-disk size for those PC's that have a speed of 3.2 and a price less than \$2000.

# Natural Joins

- The Natural join of two sets R and S is the set of pairs that agree in whatever **attributes are common** to the schemas of R and S.





ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_ID
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	-

COMPANY_ID	COMPANY_NAME	COMPANY_CITY
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

\*\* Same column came once

*Notation:  $R \bowtie S$*

- Purpose: relate rows from two tables, and
- enforce equality on all common attributes
  - eliminate one copy of common attributes

COMPANY_ID	ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_NAME	COMPANY_CITY
16	1	Chex Mix	Pcs	Akas Foods	Delhi
15	6	Cheez-It	Pcs	Jack Hill Ltd	London
15	2	BN Biscuit	Pcs	Jack Hill Ltd	London
17	3	Mighty Munch	Pcs	Foodies.	London
15	4	Pot Rice	Pcs	Jack Hill Ltd	London
18	5	Jaffa Cakes	Pcs	Order All	Boston

# Example

- Find the ID of the loan with the largest amount.
  - Hard to find the loan with the largest amount! (At least, with the tools we have so far...)
  - Much easier to find all loans that have an amount smaller than some other loan
  - Then, use set-difference to find the largest loan

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

How to find all loans with an amount smaller than some other loan?

Use Cartesian Product of *loan* with itself:

$loan \times loan$

Compare each loan's amount to all other loans

Problem: Can't distinguish between attributes of left and right *loan* relations!

Solution: Use rename operation

$loan \times \rho_{test}(loan)$

Now, right relation is named *test*

Find IDs of all loans with an amount smaller than some other loan:

$\Pi_{loan,loan\_id}(\sigma_{loan.amount < test.amount}(loan \times \rho_{test}(loan)))$

Finally, we can get our result:

$\Pi_{loan\_id}(loan) -$

$\Pi_{loan,loan\_id}(\sigma_{loan.amount < test.amount}(loan \times \rho_{test}(loan)))$

loan_id
L-421

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

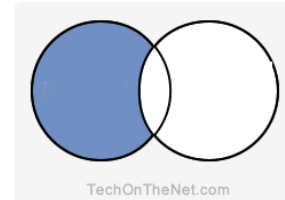
loan_id	branch_name	amount
L-421	San Francisco	7500
L-421	San Francisco	7500
L-421	San Francisco	7500
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-445	Los Angeles	2000
L-445	Los Angeles	2000
L-445	Los Angeles	2000

loan_id	branch_name	amount
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900
L-421	San Francisco	7500
L-445	Los Angeles	2000
L-437	Las Vegas	4300
L-419	Seattle	2900

# SQL: What are the names of the animals that have never been ill?

```
SELECT * FROM tables [WHERE conditions]
MINUS
SELECT * FROM tables [WHERE conditions];
```

Minus Query



**Explanation:** The MINUS query will return the records in the blue shaded area. These are the records that exist in Dataset1 and not in Dataset2. Each SELECT statement within the MINUS query must have the same number of fields in the result sets with similar data types.

```
select LESANIMAUX.NOMA from LESANIMAUX MINUS select LESMALADIES.NOMA FROM LESMALADIES
```

NOMA
1 Arthur
2 Lola
3 Tai Lung
4 Tintin
5 Tola
6 Yuan Meng

# Outer join

- Fill in missing fields with nulls

We shall consider the “natural” case first, where the join is on equated values of all attributes in common to the two relations. The *outerjoin*  $R \bowtie^o S$  is formed by starting with  $R \bowtie S$ , and adding any dangling tuples from  $R$  or  $S$ . The added tuples must be padded with a special *null* symbol,  $\perp$ , in all the attributes that they do not possess but that appear in the join result. Note that  $\perp$  is written NULL in SQL (recall Section 2.3.4).

$A$	$B$	$C$
1	2	3
4	5	6
7	8	9

(a) Relation  $U$

$B$	$C$	$D$
2	3	10
2	3	11
6	7	12

(b) Relation  $V$

$A$	$B$	$C$	$D$
1	2	3	10
1	2	3	11
4	5	6	$\perp$
7	8	9	$\perp$
$\perp$	6	7	12

(c) Result  $U \bowtie^o V$

# Left Outerjoin Example

For an example consider the tables *Employee* and *Dept* and their left outer join:

Employee

Name	EmpID	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales

Dept

DeptName	Mgr
Sales	Harriet

Name	EmpID	DeptName	Mgr
Harry	3415	Finance	null
Sally	2241	Sales	Harriet
George	3401	Finance	null
Harriet	2202	Sales	Harriet

# Example

**Exercise 5.2.1:** Here are two relations:

$$R(A, B): \{(0, 1), (2, 3), (0, 1), (2, 4), (3, 4)\}$$

$$S(B, C): \{(0, 1), (2, 4), (2, 5), (3, 4), (0, 2), (3, 4)\}$$

Compute the following: a)  $\pi_{A+B, A^2, B^2}(R)$ ; b)  $\pi_{B+1, C-1}(S)$ ; c)  $\tau_{B, A}(R)$ ;  
d)  $\tau_{B, C}(S)$ ; e)  $\delta(R)$ ; f)  $\delta(S)$ ; g)  $\gamma_{A, \text{SUM}(B)}(R)$ ; h)  $\gamma_{B, \text{AVG}(C)}(S)$ ; ! i)  $\gamma_A(R)$ ;  
! j)  $\gamma_{A, \text{MAX}(C)}(R \bowtie S)$ ; k)  $R \overset{\circ}{\bowtie}_L S$ ; l)  $R \overset{\circ}{\bowtie}_R S$ ; m)  $R \overset{\circ}{\bowtie} S$ ; n)  $R \overset{\circ}{\bowtie}_{R.B < S.B} S$ .

# Left outer join- Right outer join

There are many variants of the basic (natural) outerjoin idea. The *left outerjoin*  $R \bowtie_L S$  is like the outerjoin, but only dangling tuples of the left argument  $R$  are padded with  $\perp$  and added to the result. The *right outerjoin*  $R \bowtie_R S$  is like the outerjoin, but only the dangling tuples of the right argument  $S$  are padded with  $\perp$  and added to the result.



# Products and Joins in SQL

- SQL has a simple way to couple relations in one query: **List each relation in the FROM clause.**

**Example 6.12:** Suppose we want to know **the name of the producer of *Star Wars***. To answer this question we need the following two relations from our running example:

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

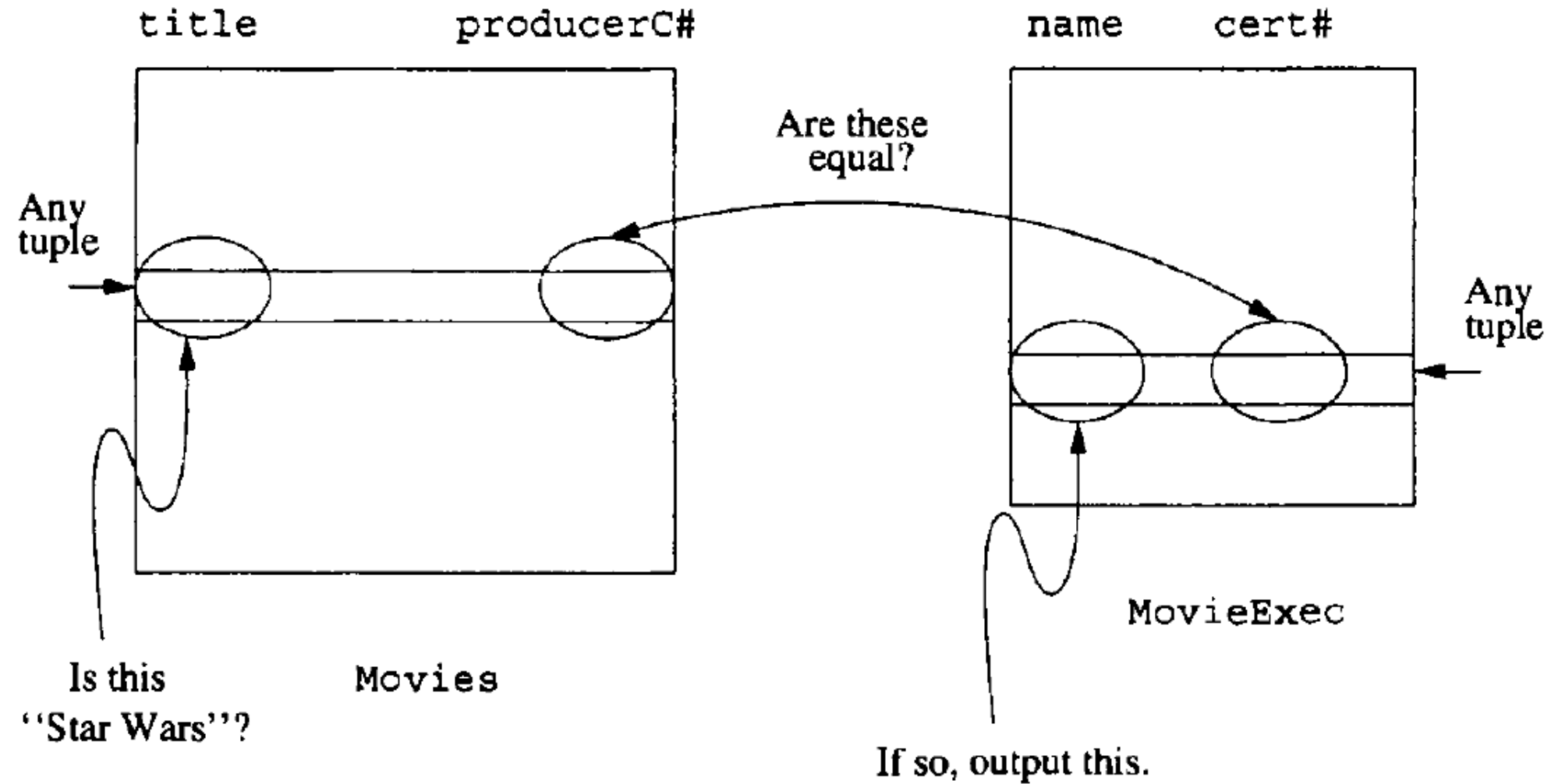
The producer certificate number is given in the **Movies** relation, so we can do a simple query on **Movies** to get this number. We could then do a second query on the relation **MovieExec** to find the name of the person with that certificate number.

However, we can phrase both these steps as one query about the pair of relations **Movies** and **MovieExec** as follows:

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

The **producerC#** attribute of the **Movies** tuple must be the same certificate number as the **cert#** attribute in the **MovieExec** tuple. That is, these two tuples must refer to the same producer.

# Products and Joins in SQL



# Products and Joins in SQL

Sometimes we ask a query involving several relations, and among these relations are two or more attributes with the same name. If so, we need a way to indicate which of these attributes is meant by a use of their shared name. SQL solves this problem by allowing us to place a relation name and a dot in front of an attribute. Thus  $R.A$  refers to the attribute  $A$  of relation  $R$ .

**Example 6.13:** The two relations

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

each have attributes `name` and `address`. Suppose we wish to find pairs consisting of a star and an executive with the same address. The following query does the job.

```
SELECT MovieStar.name, MovieExec.name
FROM MovieStar, MovieExec
WHERE MovieStar.address = MovieExec.address;
```

## Chapter 3

# Design Theory for Relational Databases

# Chapter 3

- We can examine the requirements for a database and define relations directly, without going through a high-level intermediate stage.
- In this chapter:
  - Identify the problems that are caused in some relation schemas
  - Normalization

# Functional Dependencies

Determinant  $X \rightarrow y$  Dependend

Y is determined by x

X=2 y=?

X=2 y=7

In the first relation, If I tell you the value of X  
you can find the value of Y

x	y
1	6
2	7
5	10
3	22

X=2 y=?

x	y
1	6
2	7
5	10
2	22

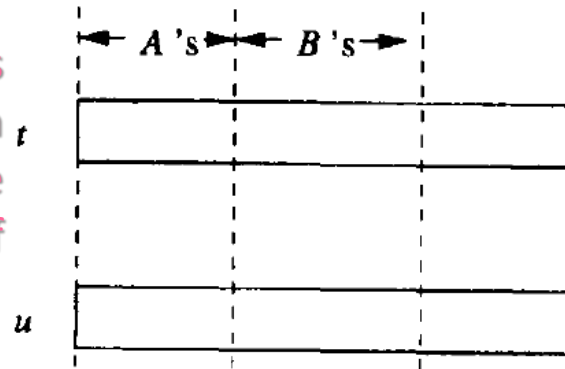
So, based on relation we can find the FD.

X and Y can be a set of attributes.

# Functional Dependencies

- If two tuples of R agree on all the attributes  $A_1, A_2, A_3, \dots, A_n$  then they must also agree on all of another list of attributes  $B_1, B_2, \dots, B_m$ .
- We write this FD formally as  $A_1 A_2 \dots A_n \rightarrow B_1 B_2 \dots B_m$  and say that  
“ $A_1, A_2, \dots, A_n$  functionally determine  $B_1, B_2, \dots, B_m$ ”

If one set of attributes in a table determines another set of attributes in the table, then the second set of attributes is said to be functionally dependent on the first set of attributes.



If  $t$  and  $u$  agree here, Then they must agree here





# Example2

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>starName</i>
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Thus, we expect that given a title and year, there is a unique movie.



Title, year  $\rightarrow$  starName

Figure 3.2: An instance of the relation `Movies1(title, year, length, genre, studioName, starName)`

As we shall see, the schema for MOVIES1 is not a good design.

Title, year  $\rightarrow$  length, genre, StudioName

This FD says that two tuples have the same value in their TITLE and Year components, the these two tuples must also have

- the same values in their Length component,
- the same values in their genre components,
- the same values in their studioName components

# Functional Dependencies

- FD says something about all possible instances of the relation, not about one of its instances.

$X \rightarrow Y$

IF  $\text{tuple}(i).x = \text{tuple}(j).x$  then  
 $\text{tuple}(i).y = \text{tuple}(j).y$

X	y
1	6
2	7
1	6
3	22

x	y
1	6
2	7
1	6
2	22



Only we need to find tuples with two equal value, then check the IF statement

If y for both of them is same so  $x \rightarrow y$  is FD.

# Keys of Relations

- We say a set of one or more attributes  $\{A_1, A_2, A_3, \dots, A_n\}$  is a key for a relation R if:
  1. Those attributes functionally determine all other attributes of a relation .  
*It is impossible for two distinct tuples of R to agree on all  $A_1, A_2, A_2, \dots, A_n$*
  2. No proper subset of  $\{A_1, A_2, A_3, \dots, A_n\}$  functionally determines all other attributes of R;  
*a key must be minimal.*

-{title, year, starname} form a key

1- two different tuples can not agree on all of title, year and starname

2-no proper subset of it functionally determines all other attributes ({title , year} is not a key)

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>starName</i>
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

**Sometimes a relation has more than one key.**

Figure 3.2: An instance of the relation `Movies1(title, year, length, genre, studioName, starName)`

## What Is “Functional” About Functional Dependencies?

$A_1 A_2 \dots A_n \rightarrow B$  is called a “functional” dependency because in principle there is a function that takes a list of values, one for each of attributes  $A_1, A_2, \dots, A_n$  and produces a unique value (or no value at all) for  $B$ . For instance, in the `Movies1` relation, we can imagine a function that takes a string like “Star Wars” and an integer like 1977 and produces the unique value of `length`, namely 124, that appears in the relation `Movies1`. However, this function is not the usual sort of function that we meet in mathematics, because there is no way to compute it from first principles. That is, we cannot perform some operations on strings like “Star Wars” and integers like 1977 and come up with the correct length. Rather, the function is only computed by lookup in the relation. We look for a tuple with the given title and year values and see what value that tuple has for `length`.

# SuperKey

- A set of attributes that contain a key is called superkey.
- Every superkey satisfies the **first condition of a key**.
- Thus every key is a superkey.
- However, some superkeys are not (minimal)keys.
- **A superkey need not satisfy the second condition :minimality.**

**Example 3.3:** In the relation of Example 3.2, there are many superkeys. Not only is the key

`{title, year, starName}`

a superkey, but any superset of this set of attributes, such as

`{title, year, starName, length, studioName}`

is a superkey. □

# Example3

!! **Exercise 3.1.3:** Suppose  $R$  is a relation with attributes  $A_1, A_2, \dots, A_n$ . As a function of  $n$ , tell how many superkeys  $R$  has, if:

- a) The only key is  $A_1$ .
- b) The only keys are  $A_1$  and  $A_2$ .
- c) The only keys are  $\{A_1, A_2\}$  and  $\{A_3, A_4\}$ .
- d) The only keys are  $\{A_1, A_2\}$  and  $\{A_1, A_3\}$ .

In general, if we have 'N' attributes with one candidate key then the number of **possible** superkeys is  $2^{(N-1)}$ .

Let a Relation  $R$  have attributes  $\{a_1, a_2, a_3, \dots, a_n\}$  and the candidate keys are "a1 a2", "a3 a4" then the possible number of super keys?

Super keys of(a1 a2) + Super keys of(a3 a4) – Super keys of(a1 a2 a3 a4)

$$\Rightarrow 2^{(N-2)} + 2^{(N-2)} - 2^{(N-4)}$$

Let a Relation  $R$  have attributes  $\{a_1, a_2, a_3, \dots, a_n\}$  and the candidate keys are "a1 a2", "a1 a3" then the possible number of

Super keys of (a1 a2) + Super keys of (a1 a3) – Super keys of(a1 a2 a3)

$$\Rightarrow 2^{(N-2)} + 2^{(N-2)} - 2^{(N-3)}$$

# Example

How many possible superkeys do we have in this example?

- {A} is a super key. (the values are not repeated)
- So , {A,B} is superkey, since A is a superkey.
- {A,C} , {A,D},{A,B,C},{A,C,D},{A,B,D},{A,B,C,D}
- Order does not matter
- Is {B}is a super key? No {C}No {D}No
- {B,C,D} No. So subset of {B,C,D} can not be a superkey
- **Answer is 8**

A	B	C	D
1	2	3	1
2	2	7	1
3	2	7	1
4	7	7	1
5	7	3	1
6	7	3	8

# Candidate Key

- Is a superkey whose **proper subset** is **not** a superkey. (minimal super key)

SK= {A},{A,B},{A,C},{A,B,C}

{B} OR {C} NO

SK={B,C}

SK= {A},{A,B},{A,C},{A,B,C}, {B,C}

proper subset :

Suppose  $X1=\{1,2,3\}$  and  $X2=\{1,2\}$

$X2$  is **subset** of  $x1$  if every member of  $X2$  must be member of  $X1$

$X2$  is **proper subset** of  $x1$

First  $x2$  is subset of  $x1$

But  $x1$  is not subset of  $x2$

{A,B,C} : WHOSE proper subset are {A,B}, {B,C},{A,C},{A}, {B}, {C} **CK=NO** SOME ARE SUPERKEYS

{A,C}: WHOSE proper subset are {A},{C} **CK=NO** SOME ARE SUPERKEYS

{A}: **CK=YES** {B,C}: WHOSE proper subset are {B} ,{C} none of its proper subset is sk **CK=YES**

A	B	C
1	6	3
2	6	5
3	1	3
4	1	5

So every CK is a SK  
But every SK is not a CK



# Rules about Functional Dependencies

- The ability to discover additional FD 's is essential when we discuss the design of good relation schemas

## 1- Reasoning about Functional Dependencies

**Example 3.4:** If we are told that a relation  $R(A, B, C)$  satisfies the FD's  $A \rightarrow B$  and  $B \rightarrow C$ , then we can deduce that  $R$  also satisfies the FD  $A \rightarrow C$ . How does that reasoning go? To prove that  $A \rightarrow C$ , we must consider two tuples of  $R$  that agree on  $A$  and prove they also agree on  $C$ .

## 2- The splitting/combining rule

We can replace an FD  $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$  by a set of FD's  $A_1A_2 \cdots A_n \rightarrow B_i$  for  $i = 1, 2, \dots, m$ . This transformation we call the *splitting rule*.

We can replace a set of FD's  $A_1A_2 \cdots A_n \rightarrow B_i$  for  $i = 1, 2, \dots, m$  by the single FD  $A_1A_2 \cdots A_n \rightarrow B_1B_2 \cdots B_m$ . We call this transformation the *combining rule*.

**Example 3.5:** In Example 3.1 the set of FD's:

```
title year → length
title year → genre
title year → studioName
```

is equivalent to the single FD:

```
title year → length genre studioName
```

that we asserted there. □

- However, there is no splitting rule for left sides

**Example 3.6:** Consider one of the FD's such as:

`title year → length`

for the relation `Movies1` in Example 3.1. If we try to split the left side into

`title → length`

`year → length`

then we get two false FD's. That is, `title` does not functionally determine `length`, since there can be several movies with the same title (e.g., *King Kong*) but of different lengths. Similarly, `year` does not functionally determine `length`, because there are certainly movies of different lengths made in any one year.

□

# Derivation rules

- $X, Y, Z$  are subsets of  $U$
- **Reflexivity**
  - if  $X \subseteq Y \subseteq U$  then  $Y \twoheadrightarrow X$
- **Augmentation**
  - if  $X \twoheadrightarrow Y$  and  $Z \subseteq U$  then  $X, Z \twoheadrightarrow Y, Z$
- **Transitivity**
  - if  $X \twoheadrightarrow Y$  and  $Y \twoheadrightarrow Z$  then  $X \twoheadrightarrow Z$
- **Pseudo - transitivity**
  - if  $X \twoheadrightarrow Y$  and  $Y, W \twoheadrightarrow Z$  then  $X, W \twoheadrightarrow Z$
- **Union**
  - if  $X \twoheadrightarrow Y$  and  $X \twoheadrightarrow Z$  then  $X \twoheadrightarrow Y, Z$
- **Decomposition**
  - if  $X \twoheadrightarrow Y$  and  $Z \subseteq Y$  then  $X \twoheadrightarrow Z$

# Anomalies

- Problems such as redundancy that occur when we try to cram too much into a single relation are called ***anomalies***:
  1. Redundancy
    - Information may be repeated unnecessarily in several tuples.
  2. Update anomalies
    - We may change information in one tuple but leave the same information unchanged in another.
  3. Deletion anomalies
    - If a set of values becomes empty, we may lose other information as a side effect.

# Examples

<i>title</i>	<i>year</i>	<i>length</i>	<i>genre</i>	<i>studioName</i>	<i>starName</i>
Star Wars	1977	124	SciFi	Fox	Carrie Fisher
Star Wars	1977	124	SciFi	Fox	Mark Hamill
Star Wars	1977	124	SciFi	Fox	Harrison Ford
Gone With the Wind	1939	231	drama	MGM	Vivien Leigh
Wayne's World	1992	95	comedy	Paramount	Dana Carvey
Wayne's World	1992	95	comedy	Paramount	Mike Meyers

Redundancy

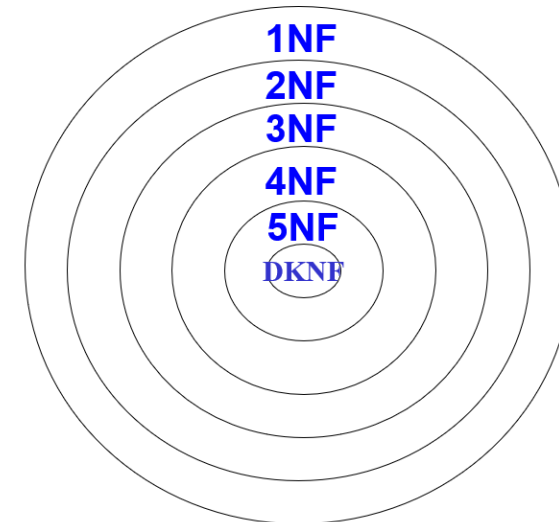
Figure 3.2: An instance of the relation `Movies1(title, year, length, genre, studioName, starName)`

- **Update anomaly:** If we found that star wars is really 125 minutes long, we might **carelessly** change the length in the first tuples but not in the second and third tuples.

# Normalization Algorithms

# Levels of Normalization

- Levels of normalization based on the amount of redundancy in the database.
  - First Normal Form (1NF)
  - Second Normal Form (2NF)
  - Third Normal Form (3NF)
  - Boyce-Codd Normal Form (BCNF)
  - Fourth Normal Form (4NF)
  - Fifth Normal Form (5NF)
  - Domain Key Normal Form (DKNF)



Each higher level is a subset of the lower level

Most databases should be 3NF or BCNF in order to avoid the database anomalies.