BASIC DATABASES

More Queries in SQL

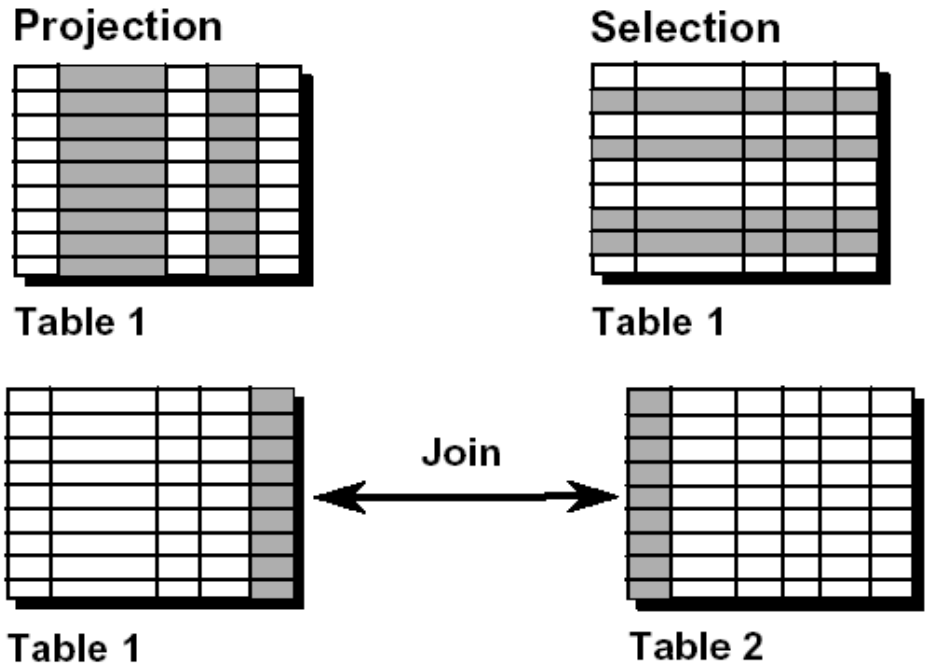
NGUYEN Hoang Ha

Email: nguyen-hoang.ha@usth.edu.vn

QUERIES INVOLVING MORE THAN ONE TABLES

Reference: Section 6.2 Jeffrey D. Ullman, Jennifer Widom: A First Course in Database Systems, Pearson, 3rd Edition (2007)

Recall: SELECT Statements →



- **Projection**: You can use the projection capability in SQL to choose the columns in a table that you want returned by your query.
- **Selection**: You can use the selection capability in SQL to choose the rows in a table that you want returned by a query (with WHERE clause)
- **Joining**: You can use the join capability in SQL to bring together data that is stored in different tables by creating a link between them.

A Cartesian Products example

EMPLOYEES (20 rows)

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
205	Higgins	110
206	Gietz	110

20 rows selected.

DEPARTMENTS (8 rows)

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
190	Contracting	1700

8 rows selected.

Cartesian product:

20x8=160 rows

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
124	50	Shipping
141	50	Shipping
206	110	Contracting

INNER JOIN

CLASS_ID	CLASS_NAME
106	Lop SE0106
107	Lop SE0107
108	Lop SE0108
201	Lop SE0201
202	Lop SE0202
203	Lop SE0203
204	Lop SE0204
205	Lop SE0205
206	Lop SE0206
207	Lop SE0207
208	Lop SE0208

STUDENT_ID	NAME	CLASS_ID
1	A	106
2	B	106
3	C	106
4	D	106
5	E	106
6	F	107
7	G	107
8	H	107
9	I	107
10	J	107
11	K	107
12	L	107
13	M	108
14	N	
15	O	

```
SELECT *
FROM student A, class B
WHERE A.class_id = B.class_id
```

```
SELECT *
FROM student A INNER JOIN class B ON A.class_id = B.class_id
```

- To determine a student's class name, you compare the value in the CLASS_ID column in the CLASS table with the CLASS_ID values in the STUDENT table.
- The relationship between the STUDENT and CLASS tables is an INNER JOIN, a.k.a *equijoin* — that is, values in the CLASS_ID column on both tables must be equal.

Express INNER JOIN in RA

$$\delta_{A.class\_id = B.class\_id}(A \times B)$$

- SELECT *
- FROM student A, class B
- WHERE A.class_id = B.class_id

Non-INNER JOIN

- A non-INNER JOIN is a join condition containing something other than an equality operator.

STUDENT_ID	MARKS
1	5
2	5
3	6
4	7
5	8
6	9
7	10
8	6
9	7
10	3
11	2

GRADE	MIN	MAX
A	8	10
B	5	8
C	0	5

```
SELECT A.student_id, B.grade
FROM student A, ranking B
WHERE A.marks >= B.MIN
      AND A.mark < B.MAX
```

Express Non-INNER JOIN with Relational Algebra?

$$\delta_{\text{A.mark >= B.min AND A.mark < B.max}}(\text{A x B})$$

- SELECT A.student_id, B.grade
- FROM student A, ranking B
- WHERE A.marks >= B.MIN
  AND A.mark < B.MAX

Outer Joins

- If a row does not satisfy a join condition, the row will not appear in the query result.
- The missing rows can be returned if an *outer join* operator is used in the join condition.

CLASS

CLASS_ID	CLASS_NAME
106	Lop 106
107	Lop 107
201	Lop 201
202	Lop 202

STUDENT

CLASS_ID	ID	NAME
106	1	A
106	2	B
107	3	C
107	4	D
	5	E
	6	F
	7	G
	8	H

Left Outer Join

- List CLASS_NAME of all the class, regardless of whether or not they have any student or not when joining CLASS and STUDENT

CLASS

CLASS_ID	CLASS_NAME
106	Lop 106
107	Lop 107
201	Lop 201
202	Lop 202

STUDENT

CLASS_ID	ID	NAME
106	1	A
106	2	B
107	3	C
107	4	D
	5	E
	6	F
	7	G
	8	H

```
SELECT A.class_name
FROM class A LEFT OUTER JOIN student B
ON A.class_id = B.class_id
```

Output of LEFT OUTER JOIN

CLASS		STUDENT		
CLASS_ID	CLASS_NAME	CLASS_ID	ID	NAME
106	Lop 106	106	1	A
106	Lop 106	106	2	B
107	Lop 107	107	3	C
107	Lop 107	107	4	D
201	Lop 201			
202	Lop 202			

The red box is on the Left

Right Outer Join

- List NAME of all the student, regardless of whether or not they have any class or not when joining CLASS and STUDENT

CLASS	
CLASS_ID	**CLASS_NAME**
106	Lop 106
107	Lop 107
201	Lop 201
202	Lop 202

STUDENT		
CLASS_ID	**ID**	**NAME**
106	1	A
106	2	B
107	3	C
107	4	D
	5	E
	6	F
	7	G
	8	H

```
SELECT B.name
FROM class A RIGHT OUTER JOIN student B
ON A.class_id = B.class_id
```

Output of RIGHT OUTER JOIN

CLASS		STUDENT		
CLASS_ID	CLASS_NAME	CLASS_ID	ID	NAME
106	Lop 106	106	1	A
106	Lop 106	106	2	B
107	Lop 107	107	3	C
107	Lop 107	107	4	D
			5	E
			6	F
			7	G
			8	H

The red box is on the Right

Full Outer Join

- List NAME of all the student and ALL classes, regardless of whether or not they have any class or not when joining CLASS and STUDENT

CLASS

CLASS_ID	CLASS_NAME
106	Lop 106
107	Lop 107
201	Lop 201
202	Lop 202

STUDENT

CLASS_ID	ID	NAME
106	1	A
106	2	B
107	3	C
107	4	D
	5	E
	6	F
	7	G
	8	H

```
SELECT B.name
FROM class A  FULL OUTER JOIN student B
ON A.class_id = B.class_id
```

Output of FULL OUTER JOIN

CLASS		STUDENT		
CLASS_ID	CLASS_NAME	CLASS_ID	ID	NAME
106	Lop 106	106	1	A
106	Lop 106	106	2	B
107	Lop 107	107	3	C
107	Lop 107	107	4	D
			5	E
			6	F
			7	G
			8	H
201	Lop 201			
202	Lop 202			

LEFT OUTER JOIN

RIGHT OUTER JOIN

Why it's called inner/outter?

- SELECT A.class_name, B.name
  FROM class A, student B
  WHERE A.class_id = B.class_id

What is the Domain Values of A.class_id?

What is the Domain Values of B.class_id?

CLASS_ID	CLASS_NAME
106	Lop SE0106
107	Lop SE0107
108	Lop SE0108
201	Lop SE0201
202	Lop SE0202
203	Lop SE0203
204	Lop SE0204
205	Lop SE0205
206	Lop SE0206
207	Lop SE0207
208	Lop SE0208

STUDENT_ID	NAME	CLASS_ID
1	A	106
2	B	106
3	C	106
4	D	106
5	E	106
6	F	107
7	G	107
8	H	107
9	I	107
10	J	107
11	K	107
12	L	107
13	M	108
14	N	
15	O	

Self Joins

- Sometimes you need to join a table to itself.

EMPLOYEE_ID	NAME	MANAGER_ID
101	A	
102	B	101
124	C	101
149	D	101
201	E	200
200	F	200
206	G	200

```
SELECT A.employee_id + ' MANAGED BY ' +
       B.employee_id
FROM employee A, employee B
WHERE A.Manager_id = B.Employee_ID
```

Cartesian Products

- A Cartesian product is formed when the join condition is omitted → All rows in the first table are joined to all rows in the second table

- To avoid a Cartesian product, always include a valid join condition in a WHERE clause.

Disambiguating attributes

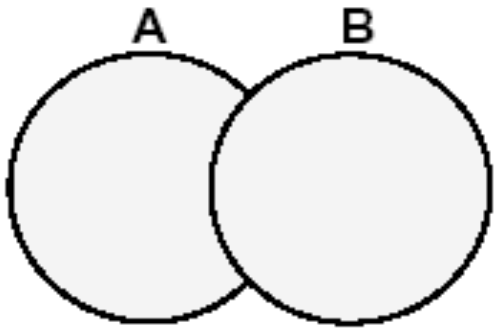
- Sometimes we ask a query involving several relations, and among these relations are two or more attributes with the same name. If so, we need a way to indicate which of these attributes is meant by a use of their shared name.
- SQL solves this problem by allowing us to place a relation name and a dot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

Disambiguating attributes example

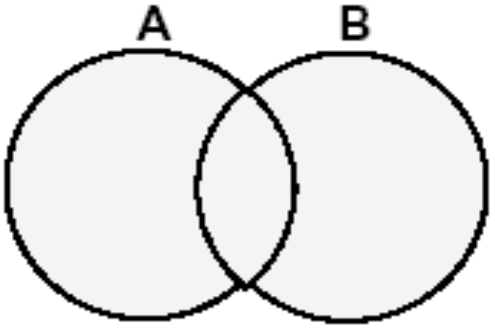
- Two relations:
  - **MovieStar (name, address, gender, birthdate)**
  - **MovieExec (name, address, cert#, netWorth)**
- Both relations have attributes "name" and "address". Look at the following query to see the way of dis-ambiguating:

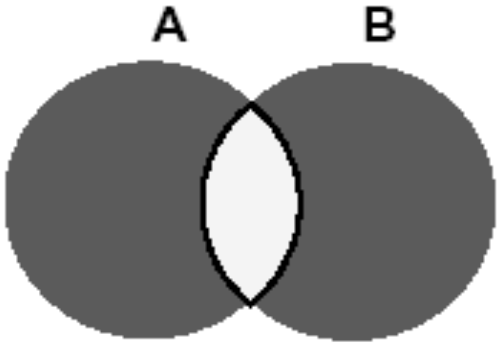
```
SELECT MovieStar.Name, MovieExec.Name
FROM MovieStar, MovieExec
WHERE MovieStar.Address = MovieExec.Address
```

SET operators

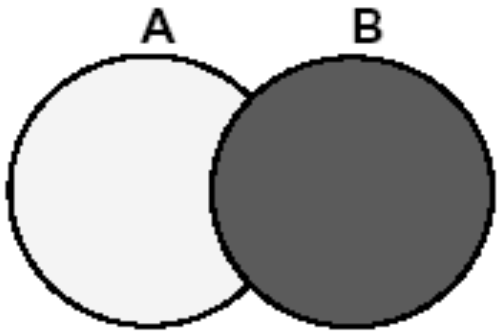


UNION/UNION ALL





INTERSECT



MINUS

UNION & UNION ALL

- The UNION operator eliminates any duplicated rows.

```
SELECT employee_id, job_id
FROM   employees
UNION
SELECT employee_id, job_id
FROM   job_history;
```

- But UNION ALL still returns duplicated rows

```
SELECT employee_id, job_id, department_id
FROM   employees
UNION ALL
SELECT employee_id, job_id, department_id
FROM   job_history
ORDER BY  employee_id;
```

INTERSECT

- {A ∩ B} = {a | a is in A and B}

```
SELECT employee_id, job_id
FROM   employees
INTERSECT
SELECT employee_id, job_id
FROM   job_history;
```

Difference

```
SELECT employee_id, job_id FROM employees
EXCEPT
SELECT employee_id, job_id FROM job_history
```

SQL Server

Oracle

```
SELECT employee_id,job_id
FROM   employees
MINUS
SELECT employee_id,job_id
FROM   job_history;
```

FULL RELATION OPERATIONS

Reference: Section 6.4 Jeffrey D. Ullman, Jennifer Widom: A First Course in Database Systems, Pearson, 3rd Edition (2007)

SELECT distinct - Eliminating duplicates

- When we do projection or select data from a bag-relation, the output may have duplicate tuples
- If we do not want duplicates in the result, then we may follow the key-word SELECT by the key-word DISTINCT

SELECT distinct - Eliminating duplicates

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Now we want to select only the distinct values from the column named "City" from the table above. We use the following SELECT statement:

```
SELECT DISTINCT City FROM Persons
```

The result-set will look like this:

City
Sandnes
Stavanger

Duplicates in Set operators

- Unlike the SELECT statement, which preserves duplicates as a default and only eliminates them when instructed to by the DISTINCT keyword, the set operators (union, intersection, difference) normally eliminate duplicates
- In order to prevent the elimination of duplicates, we must follow the operator UNION, INTERSECT or EXCEPT by the keyword ALL

Duplicates in Set operators

- The UNION, INTERSECT, EXCEPT operator eliminates any duplicated rows.

- But UNION ALL, INTERSECT ALL, EXCEPT ALL still returns duplicated rows

Grouping and Aggregation Operators

STUDENT		
CLASS_ID	ID	NAME
106	1	A
106	2	B
107	3	C
107	4	D
	5	E
	6	F
	7	G
	8	H

- Group functions (or Aggregate Functions) operate on sets of rows to give one result per group. These sets may be the whole table or the table split into groups.
- **Eg**: If we want all rows with the same CLASS_ID value will be groupped: **GROUP BY class_id**

Aggregate Operators

- **AVG**
- **COUNT**
- **MAX**
- **MIN**
- **SUM**
- **STDDEV**
- **VARIANCE**

AVG ([DISTINCT | ALL] *n*)
COUNT ({ * | [DISTINCT | ALL] *expr* })
MAX ([DISTINCT | ALL] *expr*)
MIN ([DISTINCT | ALL] *expr*)
STDDEV ([DISTINCT | ALL] *x*)
SUM ([DISTINCT | ALL] *n*)
VARIANCE ([DISTINCT | ALL] *x*)

- DISTINCT makes the function consider only nonduplicate values;
- ALL makes it consider every value including duplicates.
- The default is ALL and therefore does not need to be specified.
- All group functions ignore NULL values.

Example: AVG, SUM, MIN, MAX, COUNT

You can use AVG and SUM for numeric data.

```
SELECT AVG(salary), MAX(salary),
       MIN(salary), SUM(salary)
FROM   employees
WHERE  job_id LIKE '%REP%';
```

You can use MIN and MAX for any data type.

```
SELECT MIN(hire_date), MAX(hire_date)
FROM   employees;
```

COUNT(*) returns the number of rows in a table.

```
SELECT COUNT(*)
FROM   employees
WHERE  department_id = 50;
```

Example: Aggregate Functions and DISTINCT

Display the number of distinct department values in the EMPLOYEES table.

```
SELECT COUNT(DISTINCT department_id)
FROM   employees;
```

Grouping

- Until now, all group functions (demonstrated in above examples) have treated the table as one large group of information.
- At times, you need to divide the table of information into smaller groups. This can be done by using the **GROUP BY** clause
- The keyword **GROUP BY** is followed by a list of *grouping* attributes.

Grouping

- **Syntax: Divide rows in a table into smaller groups by using the GROUP BY clause.**

```
SELECT column, group_function (column)
FROM table
[WHERE conditions]
[GROUP BY group_by_expression]
[ORDER BY {column [ASC | DESC] ,…} ]
```

group_by_expression: specifies columns whose values determine the basis for grouping rows

Example: Aggregate Functions and GROUP BY

All columns in the `SELECT` list that are not in group functions must be in the `GROUP BY` clause.

```
SELECT department_id, AVG(salary)
FROM   employees
GROUP BY department_id;
```

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

Example: Aggregate Functions and GROUP BY

The `GROUP BY` column does not have to be in the `SELECT` list.

```
SELECT   AVG(salary)
FROM     employees
GROUP BY department_id;
```

AVG(SALARY)
4400
9500
3500
6400
10033.3333
19333.3333
10150
7000

Example: Aggregate Functions and GROUP BY

Using the GROUP BY Clause on Multiple Columns

```
SELECT   department_id dept_id, job_id, SUM(salary)
FROM     employees
GROUP BY department_id, job_id;
```

DEPT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000

Grouping: rules to remember

```
SELECT column, group_function (column)
FROM table
[WHERE conditions]
[GROUP BY group_by_expression]
[ORDER BY {column [ASC | DESC] ,…} ]
```

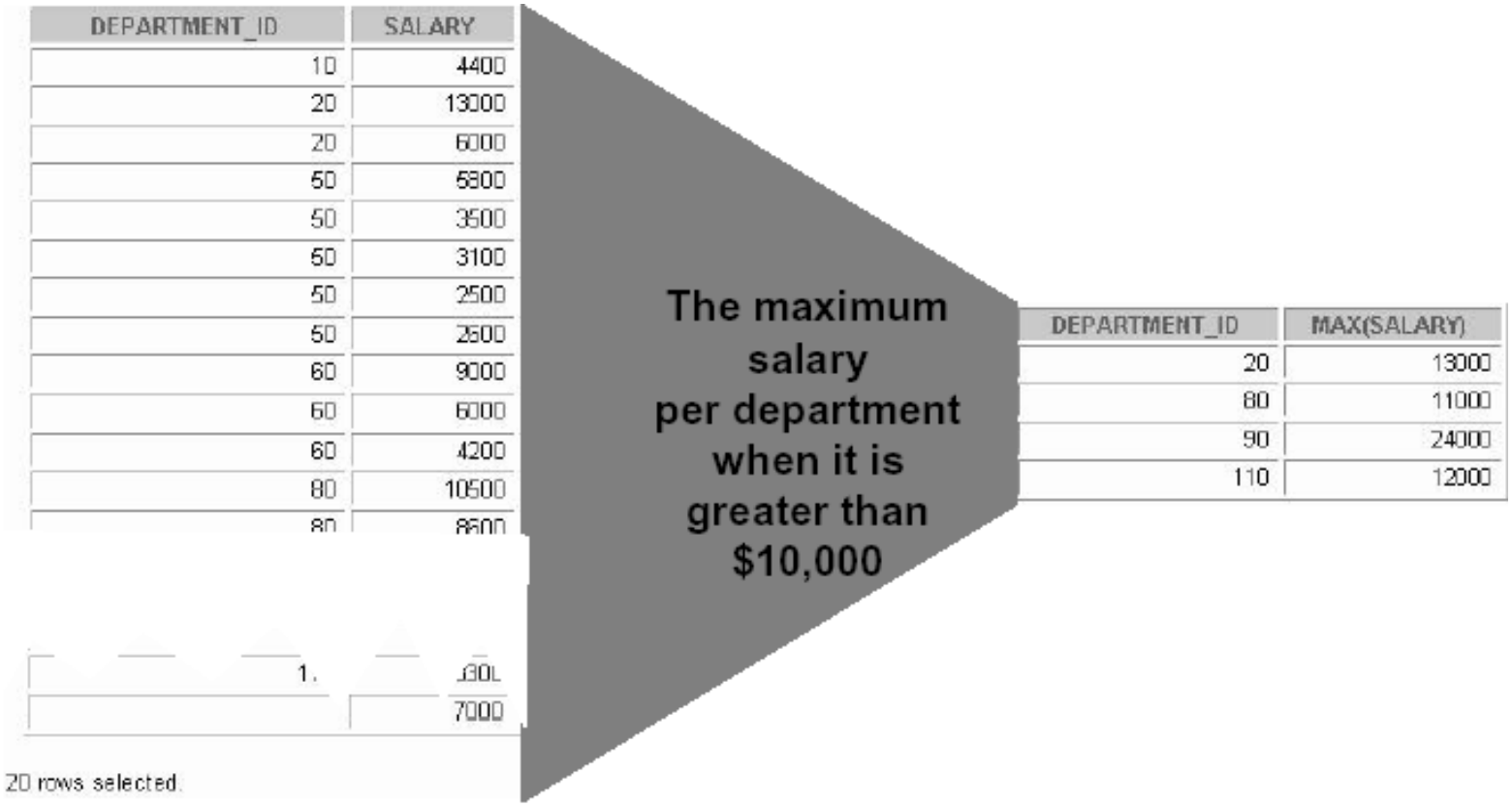
When aggregate functions are used in a select list, the select list can contain only:

- Aggregate functions.
- Grouping columns from a GROUP BY clause.
- An expression that returns the same value for every row in the result set, such as a constant

Grouping, Aggregation, and NULLs

- When tuples have nulls, there are a few rules we must remember:
  - **The value NULL is ignored in any aggregation**: it does not contribute to a SUM, AVG or COUNT of an attribute, nor can it be the minimum or maximum in its column.
    Exp: COUNT(*) is always a count of the number of tuples in a relation; but COUNT(A) is the number of tuples with non-NULL values for attribute A
  - **On the other hand, NULL is treated as an ordinary value when forming groups**: that is, we can have a group in which one or more of the grouping attributes are assigned the value NULL
  - When we perform any aggregation except COUNT over an empty bag of values, the result is NULL (the COUNT of an empty bag is 0)

HAVING clauses



- Sometimes, we want to exclude some groups from displaying result. The solution is using HAVING clause

HAVING clauses

- **Syntax**:

```
SELECT column, group_function (column)
FROM table
[WHERE conditions]
[GROUP BY group_by_expression]
[HAVING conditions]
[ORDER BY {column [ASC | DESC] ,…} ]
```

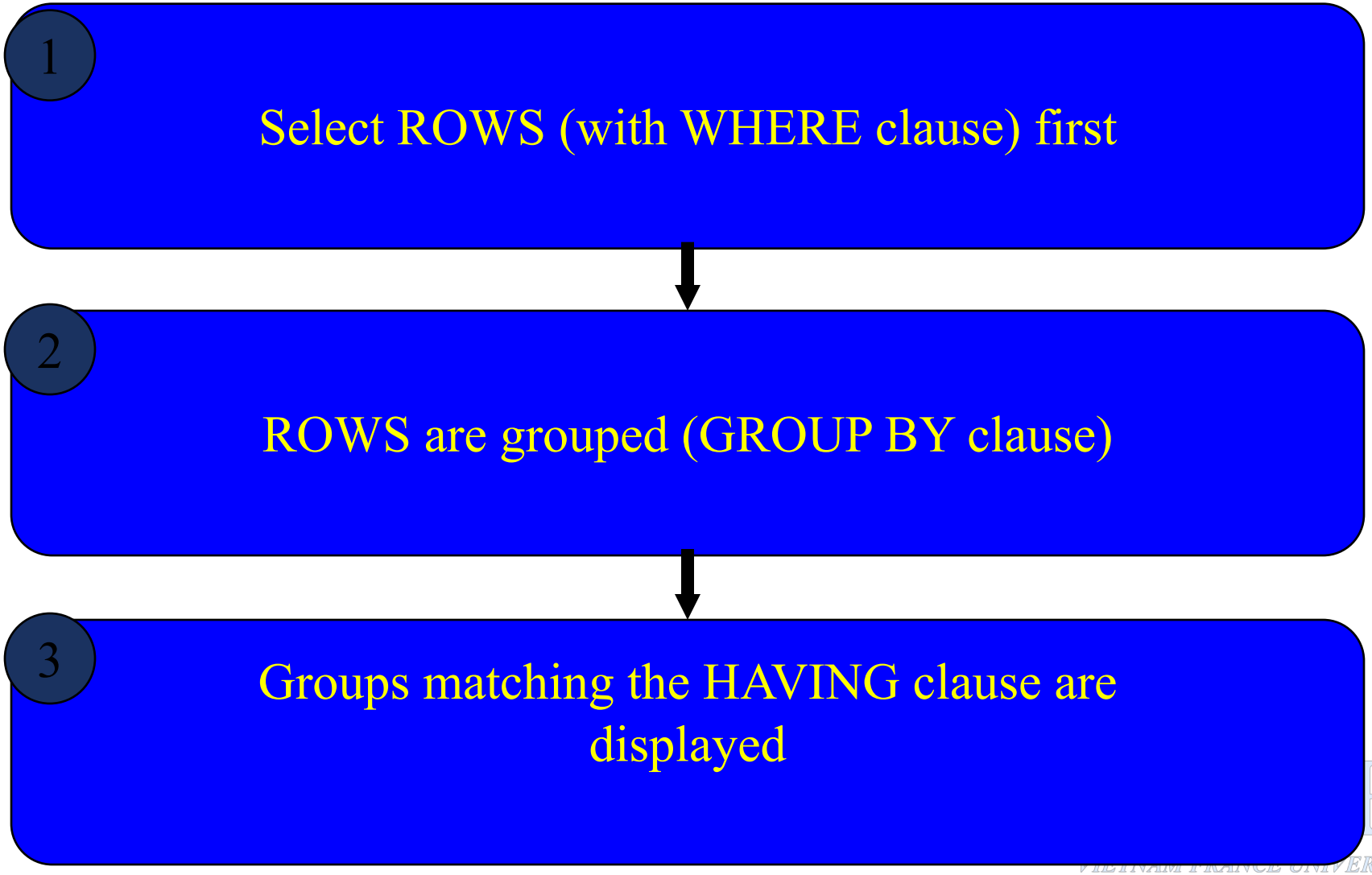
- The WHERE clause is used to restrict the rows that you select
- But the HAVING clause is used to restrict groups.

HAVING clauses

```
SELECT column, group_function (column)
FROM table
[WHERE conditions]
[GROUP BY group_by_expression]
[HAVING conditions]
[ORDER BY {column [ASC | DESC] ,…} ]
```

- Groups are formed and group functions are calculated before the HAVING clause is applied to the groups.
- In Oracle (not SQL Server), the HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because it is more logical.

HAVING clauses



Example: HAVING

```
SELECT    department_id, MAX(salary)
FROM      employees
GROUP BY department_id
HAVING    MAX(salary)>10000;
```

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

- The example displays department numbers and maximum salaries for those departments whose maximum salary is greater than $10,000.

Example: HAVING

```
SELECT   job_id, SUM(salary) PAYROLL
FROM     employees
WHERE    job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING   SUM(salary) > 13000
ORDER BY SUM(salary);
```

JOB_ID	PAYROLL
IT_PROG	19200
AD_PRES	24000
AD_VP	34000

- The example displays the job ID and total monthly salary for each job with a total payroll exceeding $13,000.

Example: HAVING

Nesting Group Functions

```
SELECT MAX(AVG(salary))
FROM   employees
GROUP BY department_id;
```

MAX(AVG(SALARY))
19333.3333

- Group functions can be nested to a depth of two. The example displays the maximum average salary.