SQL (II)
DML Query Sentences

Databases

2018-2019

Jesús Correas – jcorreas@ucm.es

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

.

DML – Data Queries

- Queries are done in SQL by means of the **`SELECT` sentence.**
- The basic syntax is:

```
SELECT C j1 ,...,C jr
FROM table1, table2,..., tablek
WHERE condition
```

- The **`SELECT` clause** specifies which columns (or expressions) will be included in the query result (it is equivalent to **projection** operation π of Relational Algebra).
- The **`FROM` clause** specifies which are the tables involved in the query (equivalent to the **cartesian product**).
- The **`WHERE` clause** is optional and specifies the selection conditions (equivalent to the **selection** operation σ).
  - If the `WHERE` clause is omitted, **all rows are retrieved.**

DML – Data Queries

- **Examples:**

Allocation

PrjCode	EmplId	Hours
PR1	27347234T	20
PR2	27347234T	25
PR3	27347234T	25

  - Select the project codes that employee 27347234T is allocated to:

```
SELECT PrjCode FROM Allocation
WHERE EmplId = '27347234T';
```

  - Select the Ids of the employees that work between 15 and 25 hours in some project, the number of hours and the project code:

```
SELECT PrjCode,EmplId,hours FROM Allocation
WHERE hours >= 15 AND hours <= 25;
```

  - **What is the result of the following query?**

```
SELECT EmplId FROM Allocation WHERE hours > 20;
```

DML – Data Queries

- SQL queries use **multisets** instead of sets: **There can produce duplicate rows.**

```
SELECT EmplId FROM Allocation WHERE hours > 20;
```

```
EmplId
-----------
27347234T
27347234T
```

- we have to use the **DISTINCT** clause to produce a set (removing duplicate rows) instead of a multiset:

```
SELECT DISTINCT EmplId FROM Allocation WHERE hours > 20;
```

```
EmplId
-----------
27347234T
```

DML – Data Queries

- If we want to obtain **all columns** from the tables included in the **`FROM`** clause, we have to use * in the **`SELECT`** clause:

```
SELECT * FROM Allocation WHERE hours > 20;
```

- The **`SELECT`** clause can include **expressions** (arithmetic, character string concatenation, function calls):

```
SELECT PrjCode || ' - ' || EmplId, Hours/8
FROM Allocation;
```

- We can change the **names of the columns** that will appear in the query result:

```
SELECT EmplId "Id del empleado" FROM distribucion;
```

- In some cases we have to execute a query that do not require any table. We can use the table **`DUAL`** to evaluate expressions.
  - It is a table with a single row and a single column (named **`DUMMY`**).
  - **Example:** **`SELECT SYSDATE, 2*3, sqrt(2) FROM DUAL;`**

DML – Query Evaluation

- The evaluation of a basic **`SELECT`** sentence can be seen as the execution of the following steps:
  1. Computation of the **cartesian product** of the tables in the FROM clause.
  2. **Deletion of the rows** that do not satisfy the condition in the **`WHERE`** clause.
  3. **Deletion of the columns** not included in the list of expressions of the **`SELECT`** clause.
  4. If **`DISTINCT`** is specified, **deletion of duplicate rows.**
- This strategy is inefficient and it is not the actual procedure used by DBMS, but it allows us to understand the meaning of `SELECT` queries.

DML – `WHERE` Clause Condition

- The condition in the **`WHERE`** where clause must be a **Boolean expression** composed of:
  - Boolean operators **`AND`**, **`OR`**, **`NOT`** and
  - simple Boolean conditions.
- **Simple Boolean conditions** are:
  - Comparison operators: **`<, >, <=, >=, =, !=`**
  - Null value checking operators: ***`atributo`*** **`IS [NOT] NULL`**
  - Membership of a given set:
    `expression` **`[NOT] IN (`*`v1,...,vn`*`)`**
  - Membership of a range:
    `expression` **`[NOT] BETWEEN`** ***`v1`*** **`AND`** ***`v2`***
  - Pattern-based character string comparison:
    `expression` **`[NOT] LIKE '`*`pattern`*`'`**
    pattern is a character string with wildcards:
    - ★ Character `_` matches any character.
    - ★ Character `%` matches a character string of any length (including the empty string).

DML – WHERE Clause Condition

- **Examples:**
  - Display the data of employees whose name start with 'Te':

    ```
    SELECT * FROM Empl WHERE name LIKE 'Te%';
    ```
  - Display the name of those employees with an "a" in the third character of their names:

    ```
    SELECT name FROM Empl WHERE name LIKE '__a%';
    ```
  - Display the data of the employees that do not have phone number stored in the DB:

    ```
    SELECT * FROM Empl WHERE phoneNr IS NULL;
    ```
  - Display the data of the employees that have their salary in a range:

    ```
    SELECT * FROM Empl
    WHERE salary BETWEEN 25000 AND 25500;
    ```
  - Display the data of the employees whose EmplId is in a given set:

    ```
    SELECT * FROM Empl WHERE EmplId IN (32,44,99);
    ```

DML – Sorting the results of a query

- The **SELECT** sentence **does not sort the resulting list of rows** unless we use an **ORDER BY** clause.
- We can specify several columns of expressions in the `ORDER BY` clause for sorting the results.
- It must be the last clause in the **SELECT** sentence.
- **Examples:**
  - Select project code, EmplId and number of hours allocated to thos employees that work more than 10 hours in those projects, sorted by the number of hours allocated.

    ```
    SELECT PrjCode, EmplId, hours FROM Allocation
    WHERE hours > 10 ORDER BY hours;
    ```
  - The same as before, but sorted in descending order:

    ```
    SELECT PrjCode, EmplId, hours FROM Allocation
    WHERE hours > 10 ORDER BY hours DESC;
    ```
  - The same as before, sorted by project code in ascending order and, in each project, sorted by number of hours in descending order:

    ```
    SELECT PrjCode, EmplId, hours FROM Allocation
    WHERE hours > 10 ORDER BY PrjCode ASC, hours DESC;
    ```

DML – Functions

- We can use expressions (arithmetic, string) that contain function calls in **`SELECT`** and **`WHERE`** clauses.
- There exists a great number of functions predefined in Oracle for **mathematical operations**, **character strings** and **date** handling. Some of them follow:

Mathematical Functions	
`ROUND(n,d)`	Rounds n to the closest value with d decimals.
`TRUNC(n,d)`	Truncates n to d of decimals.
`MOD(n1,n2)`	Returns the remainder of the integer division n1/n2.
`POWER(v,exp)`	Computes v^{exp}: the exp power of v.
`SQRT(n)`	Square root of n.
`SIGN(n)`	Returns 1 if $n > 0$, 0 if $n = 0$ and -1 if $n < 0$.
`ABS(n)`	Returns the absolute value of n.
`EXP(n)`	Returns e^n.

DML – Character String Functions

Character string functions	
LOWER(*text*)	Converts all letters in *text* to lowercase.
UPPER(*text*)	Converts all letters in *text* to uppercase.
INITCAP(*text*)	Converts the first letter of every word to uppercase.
RTRIM(*text*)	Removes blank spaces from the right of *text*.
LTRIM(*text*)	Removes blank spaces from the left of *text*.
TRIM(*text*)	Removes blank spaces from left and right of *text*.
TRIM(*c* FROM *s*)	Removes from *s* all characters that are in *c*.
SUBSTR(*s*,*n*[,*m*])	Returns *m* characters from *s* starting form the *n*-th.
LENGTH(*text*)	Returns the size of *text*.
REVERSE(*text*)	Reverses *text*.

Other character string functions:

- **INSTR(*text*,*str*[,*ini*[,*num*]])**: Searches for *str* in *text*.
- **REPLACE(*text*,*str1*[,*str2*])**: Replaces *str1* by *str2* in *text*.
- **RPAD(*text*,*max*[,*c*])**: Pads *text* with the character *c* to *max* characters. **LPAD** is the same, left-padded.

DML – NULL-related Functions

NULL-related Functions	
NVL(*v, s*)	Returns *s* if *v* is null; returns *v* otherwise.
NVL2(*v, s1, s2*)	Returns *s*1 if *v* is not null, *s*2 otherwise.
COALESCE(*exprList*)	Returns the first non-null value in *exprList*.

Ejemplos:

```
CREATE TABLE test (
  col1 VARCHAR2(1),
  col2 VARCHAR2(1),
  col3 VARCHAR2(1)
);
INSERT INTO test VALUES (NULL, 'B', 'C');
INSERT INTO test VALUES ('A', NULL, 'C');
INSERT INTO test VALUES (NULL, NULL, 'C');
INSERT INTO test VALUES ('A', 'B', 'C');
SELECT COALESCE(col1, col2, col3) FROM test;
```

DML – Date Functions

Date Functions	
SYSDATE	Returns current date and time.
ADD_MONTHS(*date*,*n*)	Adds *n* months to *date*.
MONTHS_BETWEEN(*f1*,*f2*)	Returns the number of months between *f1*,*f2*.
NEXT_DAY(*date*,*D*)	Returns the date matching the next "*D*" after *date*. *D* must be the name of a day of the week (in session's language: `'MONDAY'`, `'TUESDAY'`...)
LAST_DAY(*date*)	Returns the last day of the month in *date*.
EXTRACT(*v* FROM *date*)	Extracts the *v* component from *date*. *v* can be `day`, `month`, `year`, `minute`...
GREATEST(*f1*,*f2*,...,*fn*)	Returns the latest date in {*f1*,*f2*,...,*fn*}.
LEAST(f1,f2,...)	Returns the earliest daty in {*f1*,*f2*,...,*fn*}.

DML – Data Type Conversion

- Oracle tries to automatically convert data to match the expected type for the expression. **Examples:**

```
SELECT '5'+'3' FROM DUAL      -- The result is 8.
SELECT 5 || '3' FROM DUAL  -- The result is 53.
```

- There exist **explicit conversion functions** between text and numbers:
  - **TO_NUMBER(*textExpr*, [*fmt*,] [*nlsparams*])**
  - **TO_CHAR(*numberExpr*, [*fmt*,] [*nlsparams*])**
- *fmt* is a character string with the format for each character symbol:

9	Placeholder for a digit
0	Placeholder for a digit or zero if no digit
$	Dollar format
L	Placeholder for local currency symbol
S	Placeholder for sign
D	Placeholder for decimal symbol (decimal point in English; comma in Spanish)
G	Placeholder for thousands group separator (comma in English; point in Spanish)

DML – Data Type Conversion

- **Explicit conversion** between **text and date:**
  - **`TO_DATE(`*`textExpr`*`, [`*`fmt`*`,] [`*`nlsparams`*`])`**
  - **`TO_CHAR(`*`dateExpr`*`, [`*`fmt`*`,] [`*`nlsparams`*`])`**
- *fmt* is a character string with the format for each character symbol:

YY	Year in 2-digit format	Q	Semester
YYYY	Year in 4-digit format	WW	Week number in the year
MM	Month in 2-digit format	AM	AM indicator
MON	First three letters of month	PM	PM indicator
MONTH	Month name	HH12	Hour (1 to 12)
DY	First three letters of day of the week	HH24	Hour (0 to 23)
DAY	Day of the week	MI	Minutes (0 to 59)
D	Day of the week (1 to 7)	SS	Seconds (0 to 59)
DD	Day of the month	SSSS	Seconds from midnight
DDD	Day of the year		

- Separators between date components: `/ . , : ; '`.
- **Example:**

```
SELECT TO_CHAR(SYSDATE, 'DD/MONTH/YYYY, DAY HH:MI:SS')
FROM DUAL;    -- 07/NOVEMBER/2018, WEDNESDAY 11:35:15
```

DML – Combining Queries With Set-Theory Operations

- We can combine in SQL the results of several **SELECT** queries using set-theory operators: **UNION**, **INTERSECT**, and **MINUS**.
- The columns in all queries **must be similar: same number of columns and same types** (column names may differ).
- When using set-theory operators **duplicate rows are removed.**
  - We can obtain all rows using **UNION ALL** instead of **UNION.**
- **Examples:**

```
create table mytable (c1 integer, c2 varchar2(20));
insert into mytable values (1, 'row one');
insert into mytable values (2, 'row two');
insert into mytable values (3, 'row three');

SELECT * FROM mytable UNION SELECT 2*c1, c2 || ' 2' FROM mytable;

SELECT c1 FROM mytable INTERSECT SELECT 2*c1 FROM mytable;
```

DML – Joins on tables

- SQL provides several kinds of **joins operations on tables,** as we have seen in relational algebra.
- Join operations are specified in the **`FROM`** clause as follows:
  - ***table1* `CROSS JOIN` *table2*** **(equivalent to *table1*,*table2*)**
  - ***table1* `NATURAL JOIN` *table2***
  - ***table1* `JOIN` *table2* `USING` (*col1*,...,*colk*)**
  - ***table1* `JOIN` *table2* `ON` *joinCond***
  - ***table1* `LEFT OUTER JOIN` *table2* `ON` *joinCond***
  - ***table1* `RIGHT OUTER JOIN` *table2* `ON` *joinCond***
  - ***table1* `FULL OUTER JOIN` *table2* `ON` *joinCond***
- ***joinCond*** contains the conditions for joining `table1` and `table2`.
- The **`WHERE`** clause can contain column names from all tables in the query.
- If several columns share the same name, use ***table.col*** or table aliases (see below).
- Several joins can combine more than two tables in the same sentence.

DML – CROSS JOIN

- There are two ways to combine two tables with a cross join:

```
SELECT c1...cm FROM table1,table2 WHERE cond;
```

```
SELECT c1...cm FROM table1 CROSS JOIN table2 WHERE cond;
```

- Corresponds to the **cartesian product of Relational Algebra.**
- We can select some of the rows of the cartesian product with conditions in the `WHERE` clause.
- **Example:** Next queries are identical and produce the cartesian product `Empl`$\times$`Allocation`:

```
SELECT Name,EmplId,PrjCode FROM Empl CROSS JOIN Allocation;
```

```
SELECT Name,EmplId,PrjCode FROM Empl, Allocation;
```

DML – CROSS JOIN

- A **`CROSS JOIN`** can be used to implement the **conditional join** of relational algebra:

```
SELECT Name,EmplId,PrjCode FROM Empl, Allocation
WHERE allocation.EmplId = Empl.EmplId
```

Table **Empl**

EmplId	Name	DeptId
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Allocation**

PrjCode	EmplId	Hours
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Project**

PrjCode	ManagerId	Descr
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Dept**

DeptId	Name
SMP	Multiple Services
RH	Human Resources

- **How would be the query: "Name of employees and the description of the projects in which they work"?**

DML – Using table aliases

- We can write table aliases in the **SELECT** clause, next to the table name.
- If a table is renamed, **the alias must be used instead of the table name** for referring to a column.

```
SELECT Name, e.EmplId, a.Hours FROM Empl e, Allocation a
WHERE e.EmplId = a.EmplId;
```

- It is compulsory when joining a table twice in the same query.
- **Example:** "Name of the employees and names of the managers of the projects on which they work."

DML – Using table aliases

- We can write table aliases in the **SELECT** clause, next to the table name.
- If a table is renamed, **the alias must be used instead of the table name** for referring to a column.

```
SELECT Name, e.EmplId, a.Hours FROM Empl e, Allocation a
WHERE e.EmplId = a.EmplId;
```

- It is compulsory when joining a table twice in the same query.
- **Example:** "Name of the employees and names of the managers of the projects on which they work."

```
SELECT wrkr.Name Worker, mngr.Name Manager, p.PrjCode
FROM Empl wrkr, Allocation a, Empl mngr, Project p
WHERE a.EmplId = wrkr.EmplId
  AND p.PrjCode = a.PrjCode
  AND mngr.EmplId = p.ManagerId;
```

DML – NATURAL JOIN / JOIN USING

- The **natural join $\bowtie$ from Relational Algebra** is implemented using two alernate syntaxes:

```
SELECT c1...cm FROM table1 NATURAL JOIN table2 WHERE cond;
```

  - Performs the join comparing **all columns with the same name** in both tables using equality.
  - Does not duplicate columns with the same name (just like Relational Algebra).
  - Columns with the same name lose their table prefixes.

```
SELECT c1...cm FROM tbl1 JOIN tbl2 USING (cj,...,ck)
WHERE cond;
```

  - Performs the join comparing **the columns with the same name in the `USING` list** using equality.
  - Does not duplicate columns in **`USING`** list.
  - List columns lose their table prefixes. Remaining columns appear as in the tables they come from (using table prefix if required).

DML – NATURAL JOIN / JOIN USING

Table **Empl**

EmplId	**Name**	**DeptId**
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	**ManagerId**	**Descr**
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	**EmplId**	**Hours**
PR1	**27347234T**	20
PR3	**27347234T**	25
PR2	**27347234T**	25
PR3	**37562365F**	45
PR1	**37562365F**	10
PR1	**34126455Y**	10

Table **Dept**

DeptId	**Name**
SMP	Multiple Services
RH	Human Resources

- **Example:** "Name of the employees and code of the projects on which they work."

```
SELECT Name, PrjCode FROM Empl NATURAL JOIN Allocation;
```

DML – NATURAL JOIN / JOIN USING

Table **Empl**

EmplId	Name	DeptId
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	ManagerId	Descr
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	EmplId	Hours
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

CodDp	Nombre
SMP	Multiple Services
RH	Human Resources

- **Another example:** "Name of the employees and description of the projects on which they work."

```
SELECT Name, Descr
FROM Empl NATURAL JOIN Allocation NATURAL JOIN Project;
```

DML – NATURAL JOIN / JOIN USING

Table **Emp**

EmplId	Name	DeptId
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	ManagerId	Descr
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	EmplId	Hours
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	Name
SMP	Multiple Services
RH	Human Resources

- **Yet another example:** "Id of the employees and name of the departments they belong to."

```
SELECT Empl.Name, Dept.Name
FROM Empl JOIN Dept USING (DeptId);
```

DML – NATURAL JOIN / JOIN USING

Table **Emp**

EmplId	**Name**	**DeptId**
27347234T	Marta Sánchez	**SMP**
85647456W	Alberto San Gil	**SMP**
37562365F	María Puente	**RH**
34126455Y	Juan Panero	**SMP**

Table **Project**

PrjCode	**ManagerId**	**Descr**
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	**EmplId**	**Hours**
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	**Name**
SMP	Multiple Services
RH	Human Resources

- **Yet another example:** "Id of the employees and name of the departments they belong to."

```
SELECT Empl.Name, Dept.Name
FROM Empl JOIN Dept USING (DeptId);
```

- **Can we use `NATURAL JOIN` for this query?**

DML – JOIN ON

- The **conditional join** $\bowtie_C$ is implemented by means of **JOIN ON:**

```
SELECT c1...cm FROM table1 JOIN table2 ON joinCondition
WHERE cond;
```

  - Uses *joinCondition* to match rows from both tables.
  - *joinCondition* may include **any comparison operator:** {<, >, <=, >=, !=, =}.
  - It is **like a CROSS JOIN**, but separating **join conditions** from **selection conditions** (**WHERE** clause).
  - Unlike **NATURAL JOIN**, it duplicates columns with the same name.

- **Examples:**

```
SELECT Name, PrjCode FROM Empl JOIN Allocation
ON Empl.EmplId = Allocation.EmplId WHERE Hours > 22;

SELECT Name, PrjCode FROM Empl JOIN Project
ON Empl.EmplId = Project.ManagerId;

SELECT Name, PrjCode FROM Emp e JOIN Project p
ON e.EmplId = p.ManagerId;
```

DML – JOIN ON

Table **Empl**

EmplId	Name	DeptId
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	ManagerId	Descr
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	EmplId	Hours
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	Name
SMP	Multiple Services
RH	Human Resources

- Several kinds of joins can be **combined** (but do it with care).
- **What is the answer to this query?**

```
SELECT e.Name, PrjCode, e2.Name FROM Empl e
NATURAL JOIN Allocation
NATURAL JOIN Project p
JOIN Empl e2 ON p.MngrId=e2.EmplId;
```

DML – LEFT / RIGHT / FULL OUTER JOINs

- The join operations seen so far are **internal joins:** return those rows produced by the combination of **existing rows** in original tables.
- But we can also use **external joins:**
- **`LEFT OUTER JOIN:`** implements $\rtimes$**.**
  - Returns **all rows in first table** combined with the **matching rows of second table**, or filling with null values if there are no matching rows.
- **`RIGHT OUTER JOIN:`** implements $\ltimes$**.**
  - Returns **all rows in second table** combined with the **matching rows of first table**, or filling with null values if there are no matching rows.
- **`FULL OUTER JOIN:`** implements $\rtimes\ltimes$**.**
  - Returns **all rows in both tables,** matching them whenever possible, or filling with null values if there are no matching rows on any side.
- In all cases **we must use an `ON` or `USING` clause to include join conditions.**

DML – LEFT / RIGHT / FULL OUTER JOINs

Table **Empl**

EmplId	Name	DeptId
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	ManagerId	Descr
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	EmplId	Hours
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	Name
SMP	Multiple Services
RH	Human Resources

- **Example:** "Show the data stored for all employees, adding project information for project managers:"

```
SELECT * FROM Empl e LEFT OUTER JOIN Project p
ON p.MngrId=e.EmplId;
```

DML – Summarizing Results: Aggregation Functions

- We can use SQL to make queries where resulting rows are **grouped and summarized**.
- **Aggregation functions** are used to compute some results on **groups of rows** in a `SELECT` query:
  - `COUNT([DISTINCT]col|expr)`: returns the **number of (distinct) values** of column *col* (or expression *expr*). Rows with null value are excluded.
  - `SUM([DISTINCT]col|expr)`: returns the sum of all values of column *col* or expression *expr*.
  - `AVG([DISTINCT]col|expr)`: computes the average of the values in column *col* or expression *expr*.
  - `MAX(col|expr)`: Returns the maximum value of column/expression. `MIN` Returns the minimum value.
- In some cases, it is possible to compute those aggregated results for **distinct** values of the column/expression using `DISTINCT`.
- We can use `COUNT(*)` to count the number of rows, including duplicate rows and null values.

DML – Aggregation functions

Table **Empl**

EmplId	Name	DeptId
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	ManagerId	Descr
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	EmplId	Hours
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	Name
SMP	Multiple Services
RH	Human Resources

- **Example:** Display the total allocation of employees to projects: number of allocations, total number of hours, average allocation hours:

```
SELECT count(Hours),sum(Hours),avg(Hours) FROM Allocation;
```

- **What would be the result if we used `DISTINCT`?**
- **Another example:** Total time allocated by Employee 27347234T.

DML – Aggregations. Extended `SELECT` sentence

- Aggregation functions consider the rows of the query as a **single group** and summarize the data computing a single resulting row.
- This notion can be **extended to multiple groups:**

```
SELECT [DISTINCT] listaExpr FROM tables WHERE condW
GROUP BY GroupingColumns
HAVING GroupCondition
[ORDER BY lista];
```

- Produces as many rows as different values of ***GroupingColumns.***
- If we omit **`GROUP BY`**, all the rows in the query form a single group.
- The **`HAVING`** clause selects **which group rows** are produced by the extended query.
- **`SELECT`** and **`HAVING`** clauses can only contain expressions **available for the group rows:**
  - **Column names:** only those that are included in the **`GROUP BY`** clause.
  - Aggregation functions.

DML – Evaluation of an extended query

- The steps that should be executed for evaluating an extended **`SELECT`** sentence are as follows:
  1. Rows are generated from the tables in **`FROM`** clause and selected using the condition in the **`WHERE`** clause.
  2. Groups are set according to the values in the **`GROUP BY`** clause.
  3. The results of the aggregation functions (**`COUNT`**, **`SUM`**, **`AVG`**, ...) are computed for every group row.
  4. The group rows for which the condition in the **`HAVING`** clause holds are selected.
  5. The result is sorted according to **`ORDER BY`** clause.
- This strategy is inefficient and it is not the actual procedure used by DBMS, but it allows us to understand the meaning of extended `SELECT` queries.

DML – Extended `SELECT` sentence

Table **Empl**

EmplId	**Name**	**DeptId**
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	**ManagerId**	**Descr**
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	**EmplId**	**Hours**
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	**Name**
SMP	Multiple Services
RH	Human Resources

- **Example:** Display Employee Id and total time (in hours) allocated by those employees that work in at least 2 different projects, ordered by total time in decreasing order.

DML – Extended `SELECT` sentence

Table **Empl**

EmplId	Name	DeptId
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	ManagerId	Descr
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	EmplId	Hours
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	Name
SMP	Multiple Services
RH	Human Resources

- **Example:** Display Employee Id and total time (in hours) allocated by those employees that work in at least 2 different projects, ordered by total time in decreasing order.

```
SELECT EmplId, SUM(Hours) FROM Allocation
GROUP BY EmplId
HAVING COUNT(*)>= 2
ORDER BY SUM(Hours) DESC;
```

DML – Extended `SELECT` sentence

Table **Empl**

EmplId	**Name**	**DeptId**
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	**ManagerId**	**Descr**
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	**EmplId**	**Hours**
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	**Name**
SMP	Multiple Services
RH	Human Resources

- **Another example:** Display the project description and number of employees allocated of those projects with at least 2 employees allocated.

DML – Extended SELECT sentence

Table **Empl**

EmplId	Name	DeptId
27347234T	Marta Sánchez	SMP
85647456W	Alberto San Gil	SMP
37562365F	María Puente	RH
34126455Y	Juan Panero	SMP

Table **Project**

PrjCode	ManagerId	Descr
PR1	27347234T	Sales
PR2	37562365F	Personnel
PR3	37562365F	Logistics

Table **Allocation**

PrjCode	EmplId	Hours
PR1	27347234T	20
PR3	27347234T	25
PR2	27347234T	25
PR3	37562365F	45
PR1	37562365F	10
PR1	34126455Y	10

Table **Dept**

DeptId	Name
SMP	Multiple Services
RH	Human Resources

- **Another example:** Display the project description and number of employees allocated of those projects with at least 2 employees allocated.

```
SELECT Descr, count(*)
FROM Allocation JOIN Project USING (PrjCode)
GROUP BY PrjCode, Descr
HAVING COUNT(*)>= 2
```

DML – Nested queries

- SQL allows the use of a `SELECT` query inside another query: they are referred to as **nested queries.**
- The inner query is called **subquery.**
- Subqueries are usually included in the **`WHERE`** clause, although they can also be included in **`FROM`** or **`HAVING`** clauses.
- Subqueries in **`WHERE`** clause are used for checking multiset **membership**, (multi-)set **cardinality**, or performing **comparisons.**
- If the subquery returns **a single row with a single column,** we can use **comparison operators** **`>,<,>=,<=,!= y =.`**
- **Example:** Employees with a salary below the average salary:

```
SELECT Name, Salary FROM Empl
  WHERE Salary < (SELECT avg(Salary) FROM Empl);
```

DML – Nested queries

- We can use modified versions of comparison operators ***op*** *in* {`>,<,>=,<=,!=,=` with subqueries that **return several rows** (with a single column):
  - ***expr op ANY (subquery)*** : Returns true when the comparison operator ***op*** holds for **some rows** in the subquery results.
  - ***expr op ALL (subquery)*** : Returns true when the comparison operator ***op*** holds for **all rows** in the subquery results.
- **Example:** Employees that are allocated to any of its proyects for more hours than any employee working for Project `PR1`:

```
SELECT EmplId FROM Allocation WHERE Hours > ALL
(SELECT Hours FROM Allocation WHERE CodPr='PR1');
```

DML – Nested queries

- More subquery operators:
  - Operador ***expr* [NOT] IN (*subquery*)** checks the **membership** of *expr* to the the subquery resulting multiset.
  - **Unary** operator **EXISTS (*subquery*)** returns **true** if **subquery** produces a non-empty multiset.
  - **NOT EXISTS (*subquery*)** returns **true** if **subquery** does not return any row.
  - In these casos the subquery may return rows with more than one column.
- Nested queries can be **correlated:** It happens when the subquery **depends on the related row of the outer (main) query**.
- **Example:** Project managers that are allocated to the project they manage:

```
SELECT ManagerId FROM Project p WHERE PrjCode IN
(SELECT PrjCode FROM Allocation a
 WHERE a.EmplId=p.ManagerId);
```

DML – Correlated nested queries

- A **correlated subquery** is evaluated **for each row** of the outer (main) query.
- **Example:** Employees that do not work for any project.

```
SELECT Name, EmplId FROM Empl e WHERE NOT EXISTS
(SELECT PrjCode FROM Allocation a WHERE a.EmplId=e.EmplId);
```

  - The subquery is evaluated for each row in `Empl` table.
  - If the subquery **does not** produce any result, then the row of the outer query is included in the result of the whole query.
  - Otherwise, `NOT EXISTS` operator returns `false` and the row of the outer query is not included in the result.
- **Another example:** Employees allocated to a project more hours than the average of the employees allocated to the same project.

DML – Correlated nested queries

- A **correlated subquery** is evaluated **for each row** of the outer (main) query.
- **Example:** Employees that do not work for any project.

```
SELECT Name, EmplId FROM Empl e WHERE NOT EXISTS
(SELECT PrjCode FROM Allocation a WHERE a.EmplId=e.EmplId);
```

  - The subquery is evaluated for each row in `Empl` table.
  - If the subquery **does not** produce any result, then the row of the outer query is included in the result of the whole query.
  - Otherwise, `NOT EXISTS` operator returns `false` and the row of the outer query is not included in the result.

- **Another example:** Employees allocated to a project more hours than the average of the employees allocated to the same project.

```
SELECT e.Name, a.PrjCode, a.Hours
FROM Allocation a join Empl e on e.EmplId = a.EmplId
where a.Hours > (select AVG(a2.Hours) from Allocation a2
                 where a2.PrjCode = a.PrjCode);
```

DDL – Views

- A **view** is a **virtual table:** a relation (of the Relational Model) that is not part of the logical model but appears to the user as if it were a regular table.
- A view acts like a table to (almost) all intents and purposes.
- The DBMS just stores the definition of the view (a named SQL query) instead of its results, that **are computed every time the view is used.**
- We can define a view using the following syntax:

```
CREATE VIEW view [(columnList)] AS Query
[WITH READ ONLY | WITH CHECK OPTION];
```

  - **WITH READ ONLY:** The view does not allow data modifications.
  - **WITH CHECK OPTION:** The view allows inserting and updating rows that satisfy **query conditions.**
  - Views are updatable by default: the actual data that is changed **is the data from the underlying tables** (with some restrictions).

DDL – Views

- Names of the employees allocated to projects more hours than the average allocation.
- We can solve it creating the following views for this query:
  - EmplId allocated to projects and total amount of hours allocated:

```
CREATE VIEW EmplHours (EmplId, Hours) AS
SELECT EmplId, SUM(Hours) FROM Allocation GROUP BY
EmplId;
```

  - Name of employees and total amount of hours allocated:

```
CREATE VIEW NameHours (Name, Hours) AS
SELECT Name, Hours FROM Empl NATURAL JOIN EmplHours;
```

  - Average amount of hours allocated by employees:

```
CREATE VIEW AvgHours (average) AS
SELECT AVG(Hours) FROM NameHours;
```

- Finally, the query would be as follows:

```
SELECT Name FROM NameHours
WHERE Hours > (SELECT average FROM AvgHours);
```

DML – Updatable views

- It is possible to perform updates on views: the data in the underlying tables will be modified.
- Updatable views have some limitations:
  - All not-null columns must appear in the view definition.
  - The view query cannot contain set operators (**`UNION`**, **`MINUS`** or **`INTERSECT`**).
  - **`DISTINCT`**, **`GROUP BY`** or **`ORDER BY`** cannot be used
  - Aggregation functions cannot be used.
- Furthermore, some kinds of join-based queries may have some columns updatable, while other columns cannot be modified.