SELECT: Think of select as a function that goes through each row performs the things/calculations instructed and then prints/outputs the values.

Uses 1 based indexing

2. SUBSTR

Basic Description: Extracts a portion of a string.

- **Syntax:** `SUBSTR(string, start_position, [length])`

Standard Usage:

SQL

```
-- Start at position 2, take 3 characters
SELECT SUBSTR('Oracle', 2, 3) FROM DUAL;
-- Result: 'rac'
```

Advanced/Hidden Usage:

- **Negative Start Position:** If the start position is negative, Oracle counts backward from the *end* of the string.

SQL

```
-- Start 2 characters from the end, take all remaining
SELECT SUBSTR('Oracle', -2) FROM DUAL;
-- Result: 'le'
```

To get the last character of a string put -1 there

- **Omitting Length:** If you leave out the `length` argument, it extracts everything from the start position to the very end.

SQL

```
SELECT SUBSTR('Oracle', 3) FROM DUAL;
-- Result: 'acle'
```

3. LENGTH

Basic Description: Returns the number of characters.

- **Syntax:** `LENGTH(string)`

Standard Usage:

SQL

```
SELECT LENGTH('Database') FROM DUAL;
-- Result: 8
```

Advanced/Hidden Usage:

- **Fixed-Width Char Traps:** If you have a column defined as `CHAR(10)` and you store the word "Cat", Oracle pads it with spaces to fill 10 characters. `LENGTH` will return 10, not 3.
- **NULL Returns NULL:** `LENGTH(NULL)` returns `NULL`, not 0. This is a common bug source in PL/SQL logic.

Uses of INSTR

4. INSTR

Basic Description: Finds the numeric position of a substring.

- **Syntax:** `INSTR(string, substring, [start_position], [nth_appearance])`

Standard Usage:

```sql
SELECT INSTR('Banana', 'a') FROM DUAL;
-- Result: 2 (Finds the first 'a')
```

Advanced/Hidden Usage:

- **Finding the Nth Occurrence:** You can specify *which* match you want. This is incredibly useful for parsing data.

```sql
-- Find the 2nd occurrence of 'a', starting search from position 1
SELECT INSTR('Banana', 'a', 1, 2) FROM DUAL;
-- Result: 4
```

- **Negative Start Position (Backwards Search):** If you provide a negative start position, Oracle jumps to the end of the string and searches *backward* for the character.

```sql
-- Find the first 'a', but scanning backwards from the end
SELECT INSTR('Banana', 'a', -1) FROM DUAL;
-- Result: 6 (The last 'a')
```

- **Check Existence:** It returns 0 if the string is not found, which is often used in boolean logic (e.g., `WHERE INSTR(col, 'xyz') > 0`).

Check existence of a string and position of a char/string

5. LPAD / RPAD

Basic Description: Pads the left or right side of a string to reach a specific total length.

- **Syntax:** `LPAD(string, total_length, [pad_string])`

Standard Usage:

```sql
SELECT LPAD('123', 5, '0') FROM DUAL;
-- Result: '00123' (Great for formatting IDs)
```

Advanced/Hidden Usage:

- **Truncation Behavior:** This is the most common mistake. If the `total_length` you request is *smaller* than the original string, Oracle **cuts off** (truncates) the string to fit that length.

```sql
SELECT RPAD('Hello World', 5, '-') FROM DUAL;
-- Result: 'Hello' (It did not pad; it deleted!)
```

- **Visual Charts:** You can use `RPAD` to create simple bar charts in the console.

```sql
-- Assume salary is 5000. Div by 1000 = 5.
SELECT RPAD('#', 5, '#') FROM DUAL;
-- Result: '#####'
```

6. TRIM

Basic Description: Removes whitespace from the ends of a string.

- **Syntax:** `TRIM([ [LEADING | TRAILING | BOTH] trim_character FROM ] string)`

Standard Usage:

```SQL
SELECT TRIM('   Hello   ') FROM DUAL;
-- Result: 'Hello'
```

Advanced/Hidden Usage:

- **Removing Specific Characters:** You aren't limited to spaces. You can trim other characters, but only **one** specific character at a time.

```SQL
-- Remove '0' from the start (Leading)
SELECT TRIM(LEADING '0' FROM '000123') FROM DUAL;
-- Result: '123'
```

- **LTRIM / RTRIM Difference:** The standard `TRIM` can only remove a literal single character. If you want to remove a *set* of characters (like "remove all x, y, and z's from the left"), you must use `LTRIM` or `RTRIM`.

```SQL
-- TRIM cannot do this, but LTRIM can:
SELECT LTRIM('<>Hello', '<>') FROM DUAL;
-- Result: 'Hello'
```

7. REPLACE

Basic Description: Replaces all occurrences of a text with another text.

- **Syntax:** `REPLACE(string, search_string, [replacement_string])`

Standard Usage:

```SQL
SELECT REPLACE('Jack and Jue', 'J', 'Bl') FROM DUAL;
-- Result: 'Black and Blue'
```

Advanced/Hidden Usage:

- **Deleting Text:** If you omit the 3rd argument (the replacement string), Oracle defaults it to `NULL`, effectively deleting the search string.

```SQL
SELECT REPLACE('1-2-3-4', '-') FROM DUAL;
-- Result: '1234'
```

- **Nested Replace:** Often used to clean dirty data.

Aggregate functions, GROUP BY and HAVING

These functions take **many** values as input and return **one** single value.

Function	Description	Null Behavior	Data Types
COUNT(col)	Counts non-null rows.	Ignores NULLs	Any
COUNT(*)	Counts **all** rows.	**Counts NULLs**	Any
SUM(col)	Adds values up.	Ignores NULLs	Number
AVG(col)	Averages values.	Ignores NULLs	Number
MAX(col)	Finds highest value.	Ignores NULLs	Number, Date, String
MIN(col)	Finds lowest value.	Ignores NULLs	Number, Date, String

Export to Sheets

Pro Tip: `MIN` and `MAX` work on Dates! `MIN(hire_date)` gives you the most senior employee (earliest date).

GROUP BY important to remember for SELECT and ORDER BY

The Rule: "You can only sort by what you see."

Once you use `GROUP BY`, the original rows are gone. They have been squashed into "piles." Therefore, your `ORDER BY` clause can **only** reference:

1. The columns you used to make the groups (`GROUP BY` columns).
2. The result of an calculation (`Aggregate` functions).

You **CANNOT** order by a raw column that is buried inside the pile.

3. The "Golden Rule" of GROUP BY

This is the **#1 rule** that causes errors for beginners. Memorize this:

If a column is in the SELECT list, it MUST be in the GROUP BY clause... ...UNLESS it is inside an Aggregate Function.

Incorrect (The "Logic Error"):

SQL

```
-- ERROR: Not a GROUP BY expression
SELECT department_id, first_name, AVG(salary)
FROM employees
GROUP BY department_id;
```

- **Why it fails:** You grouped by Department. You have one "pile" for Department 90. That pile has an average salary (one number). But that pile has *three* employees (Steven, Neena, Lex). The database doesn't know which name to display next to the single average number.

When you `GROUP BY` one column, you are asking: **"Make a pile for every Department."** When you `GROUP BY` two columns, you are **NOT** making two separate sets of piles.

Instead, you are asking: **"Make a pile for every UNIQUE COMBINATION of Department and Job."**

1. The Mental Model: "Sub-Groups"

Imagine you are sorting the `EMPLOYEES` table physically.

1. **Level 1 (Department):** You toss all the employees into buckets based on their Department ID (e.g., Department 50, Department 90).
2. **Level 2 (Job):** You look **inside** the "Department 50" bucket. You notice there are Managers (`ST_MAN`) and Clerks (`ST_CLERK`).
3. **The Action:** You don't just leave them mixed in the "Dept 50" bucket. You separate them into smaller boxes **inside** that bucket.

So, your final result isn't just "How many people are in Dept 50?" It is: "How many **Clerks** are in **Dept 50**?" vs "How many **Managers** are in **Dept 50**?"

When you say `GROUP BY Column_A, Column_B`, the database engine essentially **glues** the values of those two columns together to create a temporary "ID badge" for that row.

Let's look at this visually with a tiny dataset.

1. The Raw Data (The "messy pile")

Imagine this is your `EMPLOYEES` table. Focus on Department 50.

Name	Dept_ID	Job_ID
Alice	50	ST_CLERK
Bob	50	ST_CLERK
Charlie	50	ST_MAN
David	80	SA_REP

Export to Sheets

2. Scenario A: Grouping by ONE Column

Query: `GROUP BY Dept_ID`

Oracle looks **only** at the `Dept_ID` column.

- Alice (50) → Goes into "Pile 50"
- Bob (50) → Goes into "Pile 50"
- Charlie (50) → Goes into "Pile 50"

Result: ONE group for Department 50. *(Logic: 50 is the same as 50 is the same as 50).*

GROUP BY only 1 column

3. Scenario B: Grouping by TWO Columns (The "Unique Combination")

Query: `GROUP BY Dept_ID, Job_ID`

Oracle now creates a **Composite Key**. It glues the values together to check for matches.

- **Row 1 (Alice):**
  - Dept: `50`
  - Job: `ST_CLERK`
  - **Combo:** `50 + ST_CLERK`
  - *Action:* Is there a pile for "50-Clerk"? No. **Create Group #1.**
- **Row 2 (Bob):**
  - Dept: `50`
  - Job: `ST_CLERK`
  - **Combo:** `50 + ST_CLERK`
  - *Action:* Does this match Group #1? **Yes.** Throw Bob in Group #1.
- **Row 3 (Charlie):**
  - Dept: `50`
  - Job: `ST_MAN` (Manager)
  - **Combo:** `50 + ST_MAN`
  - *Action:* Does this match Group #1? **NO.** ("50-Manager" is not "50-Clerk").
  - *Action:* **Create Group #2.**

Even though Alice, Bob, and Charlie all work in the **same department**, they ended up in **different groups** because their *combination* of Department + Job was unique.

Group	The "Unique Combination"	Count	Who is inside?
1	`50` + `ST_CLERK`	2	Alice, Bob
2	`50` + `ST_MAN`	1	Charlie
3	`80` + `SA_REP`	1	David

Export to Sheets

Why is this called a "Unique Combination"?

Because Oracle creates a new group every time it encounters a pair of (Column A + Column B) that it hasn't seen before.

- `50` and `ST_CLERK` appear together? That's one combination.
- `50` and `ST_MAN` appear together? That is a **different** combination.

GROUP BY multiple columns visualization

If you ommit group by then the whole table will form a single group and the aggregate function will give you info treating the whole table as one group

Things to remember for order by

3. Advanced Tips & Tricks for ORDER BY

Tip 1: You can Order by an Aggregate that isn't in the SELECT list

This is a weird but valid feature. You don't have to *show* the number to sort by it.

SQL

```sql
-- "List departments, sorted by how many people are in them."
-- Notice: I am NOT selecting COUNT(*) in the top line.
SELECT department_id
FROM employees
GROUP BY department_id
ORDER BY COUNT(*) DESC;
```

- **Result:** You get a list of Department IDs. The one with the most employees is at the top, even though you can't see the count number.

Tip 2: Use Column Aliases (The Best Practice)

In Oracle, `ORDER BY` is the **last** thing to execute. This means it knows the "nicknames" (Aliases) you created in the `SELECT` clause.

SQL

```sql
SELECT department_id, AVG(salary) AS avg_sal  -- Created alias here
FROM employees
GROUP BY department_id
ORDER BY avg_sal DESC;                        -- Used alias here
```

- *Note:* You generally **cannot** use aliases in the `GROUP BY` clause (e.g., `GROUP BY avg_sal` will fail), but you **can** use them in `ORDER BY`.

DISTINCT

1. What does it do?

`DISTINCT` looks at the final result set of your query and **removes any rows that are identical.**

If you have a list of 107 employees, and you ask for their `department_id`, you will get 107 numbers (many repeated). If you use `DISTINCT`, you get a list of the ~12 unique departments that actually exist.

See next pages

2. How to use it (Syntax)

Scenario A: The Single Column (Most Common)

"What are the different job titles in our company?"

SQL

```
SELECT DISTINCT job_id
FROM employees;
```

- **Without DISTINCT:** Returns 107 rows (one for every person).
- **With DISTINCT:** Returns 19 rows (just the unique job codes).

Scenario B: Multiple Columns (The "Combination" Rule)

This is where students get confused. `DISTINCT` applies to the **entire row**, not just the first column.

"Show me unique combinations of Departments and Managers."

SQL

```
SELECT DISTINCT department_id, manager_id
FROM employees;
```

- **Logic:** It glues `department_id` and `manager_id` together.
- It looks for unique **pairs**.
  - (Dept 50, Manager 100) is distinct from (Dept 50, Manager 120).
  - It will keep both.

Rule #1: You cannot selectively "Distinct" one column

This is the most common syntax error beginners make.

WRONG:

SQL

```
-- You want all names, but unique job IDs? Impossible in one SELECT.
SELECT first_name, DISTINCT job_id
FROM employees;
```

3. DISTINCT Inside Aggregates (The "Count Unique" Trick)

This is a superpower. Sometimes you don't want to *see* the list of unique items, you just want to know *how many* there are.

You can put `DISTINCT` **inside** the parenthesis of an aggregate function.

"How many different departments actually have people in them?"

SQL

```
SELECT COUNT(DISTINCT department_id)
FROM employees;
-- Result: 11
```

Compare that to:

SQL

```
SELECT COUNT(department_id)
FROM employees;
-- Result: 106 (Counts every employee assigned to a department)
```

Uses of DISCTINCT to count number of groups without using group by

Rule #2: NULLs are treated as "One Value"

If you have multiple employees with `NULL` as their commission percentage:

- `SELECT DISTINCT commission_pct...` will return **one single row** containing `NULL`.
- It treats `NULL` as a distinct category (like a "bucket" for unknowns).

Rule #4: DISTINCT vs. GROUP BY

They often produce the same output, but their purpose is different.

- **Use** `DISTINCT`: When you want to **see** the unique values (purely for display/reading).
- **Use** `GROUP BY`: When you want to **calculate** something (SUM, AVG) for each unique value.

Example:

- `SELECT DISTINCT department_id FROM employees` → Just lists them.
- `SELECT department_id FROM employees GROUP BY department_id` → Lists them (but prepares them for math).

Single quotes vs double quoutes

You are absolutely correct that **Single Quotes** are for data and **Double Quotes** are for aliases (and strict identifiers).

Where it gets confusing is that "Case Sensitivity" works differently for data vs. identifiers.

Here is the technical breakdown of what you are missing: **The Auto-Uppercase Rule**.

1. Single Quotes (Data Values) = Strict Case Sensitivity

When you use single quotes, you are telling Oracle: "This is a specific piece of text data." Oracle compares the bytes exactly as they are.

- **The Rule:** `'Apple'` ≠ `'apple'`
- **Why:** In the database, the ASCII/binary value for 'A' is different from 'a'.

Example:

SQL

```sql
-- Assume table contains one row with name = 'Smith'

SELECT * FROM users WHERE name = 'smith'; -- Returns 0 rows (No Match)
SELECT * FROM users WHERE name = 'Smith'; -- Returns 1 row (Match)
```

2. Double Quotes (Identifiers) = The "Auto-Uppercase" Trap

This is the part most people miss. Oracle identifiers (table names, column names) behave differently depending on whether you use quotes or not.

A. Without Quotes (The Default) -> Case INSENSITIVE

If you **don't** use double quotes, Oracle automatically converts everything you type into **UPPERCASE** before running it.

- **You type:** `CREATE TABLE my_table ...`
- **Oracle sees:** `CREATE TABLE MY_TABLE ...`
- **Result:** The table is stored internally as `MY_TABLE`.

Because of this, you can query it however you like, and Oracle will always convert your query to uppercase to match the stored name.

SQL

```sql
-- All of these work and point to MY_TABLE
SELECT * FROM my_table;
SELECT * FROM MY_TABLE;
SELECT * FROM MyTaBlE;
```

B. With Double Quotes -> Strict Case Preservation

If you **do** use double quotes, you force Oracle to skip the auto-uppercase step and look for *exactly* what you typed.

- **You type:** `SELECT * FROM "my_table";`
- **Oracle sees:** `SELECT * FROM my_table;` (strictly lowercase).
- **Result: ERROR** (Table or view does not exist).
- **Why:** Oracle is looking for `my_table` (lowercase), but the table is stored as `MY_TABLE` (uppercase).

Comparison Summary

Scenario	What you type	What Oracle Actually Does	Result
Data (Single Quote)	`WHERE name = 'Smith'`	Checks for binary match of `Smith`	**Match** (if data is `Smith`)
Data (Single Quote)	`WHERE name = 'smith'`	Checks for binary match of `smith`	**No Match** (if data is `Smith`)
Object (No Quote)	`FROM mytable`	Converts to `FROM MYTABLE`	**Match** (finds `MYTABLE`)
Object (No Quote)	`FROM MyTable`	Converts to `FROM MYTABLE`	**Match** (finds `MYTABLE`)
Object (Double Quote)	`FROM "mytable"`	Looks for `mytable` (lowercase)	**ERROR** (cannot find `MYTABLE`)

SET Operations

1. Structural Constraints

- **Column Count Parity:** Both queries must select the **exact same number** of columns.
  - *Valid:* `SELECT a, b FROM T1` UNION `SELECT x, y FROM T2` (2 vs 2).
  - *Invalid:* `SELECT a, b FROM T1` UNION `SELECT x FROM T2` (2 vs 1).
- **Data Type Compatibility:** Corresponding columns (1st with 1st, 2nd with 2nd) must be of the **same data type group** (e.g., Number vs. Number, Char vs. Varchar). They do not need to be the exact same type length.
  - *Valid:* `SELECT integer_col ...` UNION `SELECT float_col ...` (Both Numeric).
  - *Invalid:* `SELECT date_col ...` UNION `SELECT number_col ...` (Date vs Number).

2. Result Set Behavior

- **Header Naming:** The column names in the final result set are **always** taken from the **first** `SELECT` statement. Aliases in the second query are ignored.
- **Duplicate Handling:**
  - `UNION`, `INTERSECT`, `MINUS`: Automatically eliminate duplicate rows.
  - `UNION ALL`: Preserves all duplicates (and is faster because it skips the sort/deduplication step).
- **NULL Handling:** Unlike standard SQL comparisons (where `NULL = NULL` is False), set operators treat `NULL` values as **identical**.
  - If Query A returns `(NULL, 10)` and Query B returns `(NULL, 10)`, `UNION` will squash them into a single row.

3. Sorting and Ordering

- **Global Sort:** The `ORDER BY` clause can appear **only once** at the very end of the compound query.
- **Reference Scope:** The `ORDER BY` must reference columns by the names defined in the **first** query, or by column position number (e.g., `ORDER BY 2`).
  - *Wrong:* `SELECT a FROM T1 ORDER BY a UNION SELECT b FROM T2`
  - *Right:* `SELECT a FROM T1 UNION SELECT b FROM T2 ORDER BY a`

5. Execution Order

- **Default Priority:** Oracle processes set operators from **Top to Bottom** (Left to Right).
  - Standard SQL often gives `INTERSECT` higher precedence than `UNION`, but Oracle strictly follows the query order unless parentheses are used.
- **Parentheses:** Use `( )` to explicitly control execution order.
  - $$A \cup (B \cap C) \neq (A \cup B) \cap C$$

6. Code Example: The "All-in-One" Syntax

SQL

```sql
SELECT employee_id, job_id, salary   -- [Rules: Col Names come from here]
FROM employees
WHERE department_id = 90

UNION ALL                            -- [Rules: Keeps duplicates]

SELECT employee_id, job_id, 0        -- [Rules: '0' matches 'salary' type]
FROM retired_employees               -- [Rules: 3 columns vs 3 columns]

MINUS                                -- [Rules: Executed after the UNION]

SELECT employee_id, job_id, salary
FROM blacklisted_employees

ORDER BY 3 DESC;                     -- [Rules: Sorts final result by 3rd col]
```

In SQL, the number 3 in ORDER BY 3 uses Positional Notation. It tells the database to sort the results based on the 3rd column listed in your SELECT statement.

Some practice with ai (learning new concept with problems)

Problem: For each employee, display their Last Name and a calculated column named VOWEL_COUNT. Filter the results to show only those employees whose Last Name contains exactly 2 instances of the letter 'a' (case-insensitive) and ends with a consonant. Sort the list by VOWEL_COUNT descending.

```sql
SELECT LAST_NAME,
       (
        LENGTH(LAST_NAME) - LENGTH(TRANSLATE(LOWER(LAST_NAME), '@aeiou', '@'))
        ) AS VOWEL_COUNT
FROM EMPLOYEES
WHERE
    -- Fix: Replace 'a' with empty string '' to reduce length
    (LENGTH(LAST_NAME) - LENGTH(REPLACE(LOWER(LAST_NAME), 'a', ''))) = 2
    AND
    SUBSTR(LOWER(LAST_NAME), -1) NOT IN ('a', 'e', 'i', 'o', 'u')
ORDER BY VOWEL_COUNT DESC;
```

Function usage here-
Translate :
Syntax: TRANSLATE(string, from_chars, to_chars)
TRANSLATE('I like cats', 'cats', 'dogs')
It looks at every character in the input. If a character appears in from_chars, it swaps it with the character at the same index in to_chars.Example: Swap 'c' to 'd', 'a' to 'o', 't' to 'g', 's' to 's'.

on the above solved question TRANSLATE is used to count the number of vowels in a clever way.
from chars contrain all the vowels but the first character is a character that won't apperar on any input string.
and in the 'to_chars' parameter we are writing only that special character (eg: @). So all the @ will be replaced by @ but there are no characters in to_chars to replace a,e,i,o,u so they get deleted.

TRANSLATE is different from REPLACE because
It searches for the exact sequence search_str. If found, it swaps that entire block with replace_str.
But TRANSLATE works on individual character with surgical precision.

So we can replace all the a with a null string to cut out all the a from the string.We are using replace here as there is only one character here.We could've also used translate here like
TRANSLATE(LOWER(LAST_NAME), '@a', '@')

SUBSTR: is used to get the substring. -1 is used to get the last character of the string.

IMPORTANT!!!! you can put null ('') in the third parameter of replace but you can't put null on the third parameter of TRANSLATE. weird rule

Question 2: Advanced Date Arithmetic & Leap Years

Concept: Precise date extraction and boolean logic with dates.

Context: Expands on the "hired in March/Sept" and "600 days" logic.

Problem: Display the `EMPLOYEE_ID`, `HIRE_DATE`, and a column `HIRE_QUARTER` (Format: 'Q1', 'Q2'...). Filter to show only employees who meet **all** the following criteria:

1. Hired in a **Leap Year**.
2. Hired in the **second half of the month** (Day 16 or later).
3. Have worked for more than **8,000 days** as of today (use `SYSDATE`).

Sort by `HIRE_DATE` ascending.

```sql
SELECT
    EMPLOYEE_ID,
    HIRE_DATE,
    'Q' || TO_CHAR(HIRE_DATE,'Q') as hire_quarter
FROM EMPLOYEES

WHERE
    MOD(EXTRACT(YEAR FROM HIRE_DATE),4 )=0
    -- MOD(TO_NUMBER(TO_CHAR(HIRE_DATE, 'YYYY')), 4) = 0
    AND EXTRACT(DAY FROM HIRE_DATE)>15
    --  TO_NUMBER(TO_CHAR(HIRE_DATE,'DD'))>15

    AND (SYSDATE-HIRE_DATE)>8000
ORDER BY HIRE_DATE ASC
```

we can use either EXTRACT or TO_CHAR to get day,month or year from dates.
Buet EXTRACT gives the number form while TO_CHAR gives varchar which needs to be converted to number using TO_NUMBER if I want to do any number things.

Problem: Display FIRST_NAME, HIRE_DATE, and the position of the character 'e'. Filter the list to show only employees where the letter 'e' (case-insensitive) appears at least twice in their first name, but the second occurrence of 'e' is at index 5 or later. Sort by HIRE_DATE descending.

```sql
SELECT FIRST_NAME,
       HIRE_DATE,
       INSTR(LOWER(FIRST_NAME), 'e', 1, 2) AS SECOND_E_POS
FROM EMPLOYEES
WHERE
      -- INSTR(string, substring, start_position, occurrence)
      -- Find position of the 2nd 'e', starting search from char 1.
      -- If result >= 5, it exists and is at index 5+.
      INSTR(LOWER(FIRST_NAME), 'e', 1, 2) >= 5
ORDER BY HIRE_DATE DESC;
```

We don't need to explicitly check if FIRST_NAME has more than 2 e or not. Cause INSTR is doing that for us.Cause a row is only included if the second e is at index 5 or later

Problem: Problem: Calculate a "Performance Score" for each employee using this formula: Score = SALARY + (10,000 extra points IF they have a commission percentage).

```sql
SELECT LAST_NAME,
    SALARY,
    COMMISSION_PCT,
    SALARY + NVL2(COMMISSION_PCT,10000,0) as PERFORMANCE_SCORE
FROM EMPLOYEES
```

NVL2 Function tutorial:

`NVL2` is a ternary operator in Oracle SQL that determines the output based on whether an expression is null or not null. It is more flexible than the standard `NVL` because it allows you to transform the value even if it is **not** null.

Syntax

$$NVL2(expr1, expr2, expr3)$$

1. `expr1` **(The Check):** The value being inspected.
2. `expr2` **(If NOT NULL):** The value returned if `expr1` contains data.
3. `expr3` **(If NULL):** The value returned if `expr1` is null.

However we can solve this by only using the NVL with a clever trick :
NVL(COMMISSION_PCT / COMMISSION_PCT, 0) * 10000
If COMMISSION_PCT is 0.2: 0.2 / 0.2 = 1 -> 1 * 10000 = 10000$.
If COMMISSION_PCT is NULL: The expression becomes NULL. NVL(NULL, 0) turns it to 0.

Problem 4:

Write a single query to find the total number of employees who are charged with commissions and who are not. The Indicator here is boolean that shows 0 for not being charged and 1 for being charged. Your output should exactly match the following:

IS_COMMISSION_CHARGED?	EMPLOYEE_COUNT
0 (NO)	72
1 (YES)	35

```sql
SELECT NVL2(commission_pct, '1 (YES)', '0 (NO)') AS "IS_COMMISSION_CHARGED?",
       COUNT(*) AS EMPLOYEE_COUNT
FROM employees
GROUP BY NVL2(commission_pct, '1 (YES)', '0 (NO)');
```

Problem 2:

Create a dept email address for each of the employees in the EMPLOYEES table. Assume that the domain is ".buet.ac.bd". Remember, the company doesn't want to have any space(' ') or underscore('_') in the email address. And everything should be in lowercase. Show the results for employees who have spaces in their first or last name.
Example output:

FIRST_NAME	LAST_NAME	JOB_ID	DEPT_MAIL
Lex	De Haan	AD_VP	lex.dehaan@advp.buet.ac.bd
Jose Manuel	Urman	FI_ACCOUNT	josemanuel.urman@fiaccount.buet.ac.bd

Solution:

```sql
select first_name, last_name, job_id,
    lower(replace(replace(
      (first_name || '.' || last_name || '@' || job_id || '.buet.ac.bd'),
    ' ',''),'_','')) as dept_mail
from hr.employees
where instr(last_name,' ') <> 0 or instr(first_name,' ') <> 0;
```

Problem 2:

The conventional method for creating an email is to combine the initial letter from the First Name field (If it has space then the first character after space too) with the entire Last Name field (without space), both in uppercase, to form the EMAIL field. Nevertheless, some employees might not have their entire last name included in the email. Find employees whose emails don't include their entire last names. Example output:

FIRST_NAME	LAST_NAME	EMAIL
Mozhe	Atkinson	MATKINSO
David	Bernstein	DBERNSTE
...	...	...
Nandita	Sarchand	NSARCHAN
Martha	Sullivan	MSULLIVA

Solution:

```sql
select first_name, last_name, email
from hr.employees
where upper(email) not like '%'||upper(replace(last_name,' ',''))||'%';
```

Problem: For employees hired in the years 1997, 1998, or 1999: Group them by the Month of Hire (e.g., 'January', 'February'). Display the Month Name and the Total Salary of employees hired in that month. Show only months where the Average Salary is exactly divisible by 100.Sort explicitly by the Month Number (1-12), not the Month Name (A-Z).

```sql
SELECT TO_CHAR(HIRE_DATE, 'Month') AS HIRE_MONTH_NAME,
       SUM(SALARY) AS TOTAL_SAL
FROM EMPLOYEES
WHERE TO_CHAR(HIRE_DATE, 'YYYY') IN ('1997', '1998', '1999')
GROUP BY TO_CHAR(HIRE_DATE, 'Month'), TO_CHAR(HIRE_DATE, 'MM')
HAVING MOD(AVG(SALARY), 100) = 0
ORDER BY TO_CHAR(HIRE_DATE, 'MM');
```

TO_CHAR(HIRE_DATE, 'Month') gives month name like January
TO_CHAR(HIRE_DATE, 'MM') gives month number as varchar like '01'

Problem:

Problem: Display the `LAST_NAME` and a generated `SECURITY_CODE`. The code format is:

1. Take the first 3 letters of `LAST_NAME`.
2. Pad it on the **left** with `*` until it is 10 characters long.
3. Pad that result on the **right** with `#` until it is 15 characters long.
4. Finally, remove any `*` from the **left** side only (using `LTRIM`).

Sort by the length of the final `SECURITY_CODE` descending.

Solution:

SQL

```sql
SELECT LAST_NAME,
       LTRIM(RPAD(LPAD(SUBSTR(LAST_NAME, 1, 3), 10, '*'), 15, '#'), '*') AS SECURITY_COD
FROM EMPLOYEES
ORDER BY LENGTH(SECURITY_CODE) DESC;
```

- **Why this is hard:** It requires visualizing the "onion layers" of functions:
  1. `SUBSTR('King', 1, 3)` -> 'Kin'
  2. `LPAD('Kin', 10, '*')` -> '*******Kin'
  3. `RPAD('*******Kin', 15, '#')` -> '*******Kin#####'
  4. `LTRIM(..., '*')` -> 'Kin#####'
  - *Result:* This tests if you understand that `LTRIM` removes the characters you just added with `LPAD`.

How to find years, months and days between two dates?

MONTS_BETWEEN and general date substraction can give fraction so TRUNC is used

```sql
SELECT employee_id,
       hire_date,
       -- 1. Calculate Years
       TRUNC(MONTHS_BETWEEN(SYSDATE, hire_date) / 12) AS years,
       -- 2. Calculate Months
       TRUNC(MOD(MONTHS_BETWEEN(SYSDATE, hire_date), 12)) AS months,
       -- 3. Calculate Remaining Days
       TRUNC(SYSDATE - ADD_MONTHS(hire_date, TRUNC(MONTHS_BETWEEN(SYSDATE, hire_date)))) AS days
FROM employees
ORDER BY years DESC, months DESC;
```

MONTHS_BETWEEN: function to get the correct number of months between two dates handles all edge cases.

ADD_MONTHS: ADD_MONTHS(D, n) = D + n * Months

The `ADD_MONTHS` function returns a date with the same time component as the input date, shifted forward or backward by a specified integer number of months.

Syntax: `ADD_MONTHS(date_expression, integer)`

Operational Rules

1. **Standard Addition:** If the target month has the same day index as the source month, that day is returned.
   - Example: Jan 15 +1 Month → Feb 15.
2. **Last-Day Protocol (The "Sticky" Rule):** If the input date is the **last day of the month**, the result will always be the **last day of the target month**, regardless of the number of days in that month.
   - Example: Feb 28 (Non-leap) +1 Month → Mar 31 (Not Mar 28).
3. **Day Overflow:** If the input date is *not* the last day, but the day index does not exist in the target month (e.g., Jan 31 +1 Month), the function returns the **last valid day** of the target month.
   - Example: Jan 31 +1 Month → Feb 28 (or 29 in leap years).
4. **Time Preservation:** The time component (hours, minutes, seconds) remains unchanged.

Problem 3:

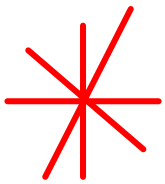
If the company begins offering a 2% extra commission to employees with over 15 years of service (and no commission for others), how much total commission will the company need to pay to its employees based on the EMPLOYEES table? For example, If an employee has worked no more than 15 years, he gets 0% commission. Otherwise, if he gets 3% commission, now he'll get 5% commission; and if he gets a null commission he'll get 2% commission now.

Solution:

```
select sum(salary*(NVL(COMMISSION_PCT, 0)+0.02)) as TOTAL_COMMISSION_NOW
from hr.employees
where (sysdate-hire_date)/365 > 15;
```

This solution is technically wrong as it doesn't account for leap year.
The correct solution is:

WHERE MONTHS_BETWEEN(SYSDATE, hire_date) / 12 > 15;



Find the total number of employees hired in each triennial period (every three years, for example: 2001-2003) for each job type. Sort them **in descending chronological order**. Example output:

TRIENNIAL_PERIOD	JOB_ID	EMPLOYEES_HIRED
2007-2009	FI_ACCOUNT	1
2007-2009	IT_PROG	2
...	...	...
2001-2003	ST_CLERK	2
2001-2003	ST_MAN	1

Solution:

```
SELECT
    ((TRUNC(TO_CHAR(HIRE_DATE, 'yyyy')/3) * 3)) || '-' ||
(((TRUNC(TO_CHAR(HIRE_DATE, 'yyyy')/3)+1) * 3)-1) AS TRIENNIAL_PERIOD,
    JOB_ID,
    COUNT(EMPLOYEE_ID) AS EMPLOYEES_HIRED
FROM
    HR.EMPLOYEES E
GROUP BY
    TRUNC(TO_CHAR(HIRE_DATE, 'yyyy')/3),
    JOB_ID
ORDER BY TRUNC(TO_CHAR(HIRE_DATE, 'yyyy')/3) DESC;
```

Problem 2:

In our HR.EMPLOYEES table, it has a column called PHONE_NUMBER which has numbers separated with dots. For example

PHONE_NUMBER
515.123.4567
515.123.4568

Assume that the first part of the phone number before the "." indicates country code, and the second part of the phone number between the first two "."'s indicate the region code.
Now, extract the country code and region code from the phone numbers. Example output:

PHONE_NUMBER	COUNTRY_CODE	REGION_CODE
515.123.4567	515	123
515.123.4568	515	123
...	...	...
011.44.1344.619268	011	44
011.44.1344.429018	011	44

```
SELECT PHONE_NUMBER,
       SUBSTR(PHONE_NUMBER,1,INSTR(PHONE_NUMBER,'.')-1) as country_code,
       SUBSTR(PHONE_NUMBER,INSTR(PHONE_NUMBER,'.')+1,
              INSTR(PHONE_NUMBER,'.',1,2)-INSTR(PHONE_NUMBER,'.')-1) as REGION_CODE
FROM EMPLOYEES
```

The Oracle `INSTR` function locates the starting position of a specific substring within a larger string. It returns an integer representing the position of the first character of the substring found. If the substring is not present, the function returns `0`. By default, the search is case-sensitive.

The basic syntax is `INSTR(string, substring [, position [, occurrence ] ])`, where `string` is the text to search, and `substring` is the text to find. The optional `position` argument specifies where to start the search (default is 1, a negative value searches backward from the end). The optional `occurrence` argument indicates which instance of the `substring` to find (default is 1 for the first occurrence).