select col1|| ' ' ||col2 as col_name from table_name order by col1 desc col2 asc;

—------------------null handling-------------------------
CREATE TABLE sort_demo(
	id serial NOT NULL primary key,
	num INT
);
INSERT INTO sort_demo(num)
VALUES(1),(2),(3),(null);

SELECT num FROM sort_demo ORDER BY num NULLS FIRST;
—--

SELECT DISTINCT col1, col2 FROM table; //remove duplicates
SELECT DISTINCT ON(col1) col1, col2 FROM table; //remove duplicates considering only col1

SELECT last_name, first_name FROM customer WHERE first_name = 'Juntak' AND last_name = 'Lee';

SELECT first_name, last_name FROM customer WHERE first_name IN ('Ann','Anne','Annie');

SELECT first_name, LENGTH(first_name) name_length FROM customer WHERE first_name LIKE 'A%' AND LENGTH(first_name) BETWEEN 3 AND 5 ORDER BY name_length;

SELECT col1 FROM table_name LIMIT row_count OFFSET row_to_skip;

memo: The FETCH clause is functionally equivalent to the LIMIT clause. If you plan to make your application compatible with other database systems, you should use the FETCH clause because it follows the standard SQL.

SELECT film_id, title FROM film ORDER BY title OFFSET 5 ROWS FETCH FIRST 5 ROW ONLY;

```
SELECT
	customer_id,
	first_name,
	last_name
FROM
	customer
WHERE
	customer_id IN (
		SELECT customer_id
		FROM rental
		WHERE CAST (return_date AS DATE) = '2005-05-27'
	);
```

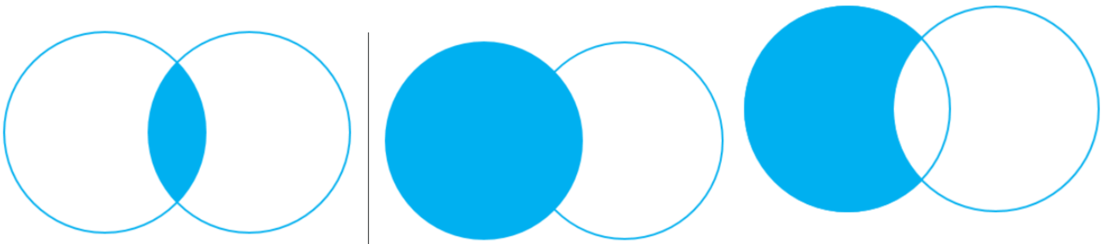
SELECT customer_id, payment_id, amount FROM payment WHERE amount NOT BETWEEN 8 AND 9;

SELECT customer_id, payment_id, amount, payment_date FROM payment WHERE payment_date BETWEEN '2007-02-07' AND '2007-02-15';

// 'foo' ILIKE '_O_',   ← true, note that ILIKE is case-insensitively unlike LIKE

SELECT id, first_name, last_name, email, phone FROM contacts WHERE phone **IS NOT** NULL;

JOIN



INNER JOIN | LEFT OUTER JOIN | LEFT OUTER JOIN – only rows from the left table

SELECT a, fruit_a, b, fruit_b FROM basket_a INNER JOIN basket_b ON fruit_a = fruit_b;
//matching the values in the `fruit_a` and `fruit_b` columns

SELECT a, fruit_a, b, fruit_b FROM basket_a LEFT JOIN basket_b ON fruit_a = fruit_b;

SELECT a, fruit_a, b, fruit_b FROM basket_a LEFT JOIN basket_b ON fruit_a = fruit_b WHERE b IS NULL;

RIGHT OUTER JOIN – only rows from the right table | FULL OUTER JOIN | FULL OUTER JOIN – only rows unique to both tables

SELECT a, fruit_a, b, fruit_b FROM basket_a RIGHT JOIN basket_b ON fruit_a = fruit_b WHERE a IS NULL;

SELECT a, fruit_a, b, fruit_b FROM basket_a FULL OUTER JOIN basket_b ON fruit_a = fruit_b;

SELECT a, fruit_a, b, fruit_b FROM basket_a FULL JOIN basket_b ON fruit_a = fruit_b WHERE a IS NULL OR b IS NULL;

a_very_long_table_name (AS) c
↓
SELECT c.customer_id, first_name, amount, payment_date FROM customer c INNER JOIN payment p ON p.customer_id = c.customer_id ORDER BY payment_date DESC;

// When you join a table to itself (a.k.a self-join), you need to use table aliases. This is because referencing the same table multiple times within a query results in an error.
↓
SELECT e.first_name employee, m .first_name manager FROM employee e INNER JOIN employee m ON m.employee_id = e.manager_id ORDER BY manager;

//join three tables
SELECT c.customer_id, c.first_name customer_first_name, c.last_name customer_last_name, s.first_name staff_first_name, s.last_name staff_last_name, amount, payment_date FROM customer c INNER JOIN payment p ON p.customer_id = c.customer_id INNER JOIN staff s ON p.staff_id = s.staff_id ORDER BY payment_date;

TODO: Section 3: Self join, Natural Join, Cross Join, Full Outer Join

// The GROUP BY clause divides the rows returned from the SELECT statement into groups. For each group, you can apply an aggregate function e.g., SUM() to calculate the sum of items or COUNT() to get the number of items in the groups.

SELECT customer_id, SUM (amount) FROM payment GROUP BY customer_id ORDER BY SUM (amount) DESC;

```
SELECT
	first_name || ' ' || last_name as full_name,
	count (amount) amount
FROM
	payment
INNER JOIN customer USING (customer_id)
GROUP BY
	full_name
ORDER BY amount DESC;
```

SELECT DATE(payment_date) paid_date, SUM(amount) sum FROM payment GROUP BY DATE(payment_date);

//The HAVING clause specifies a search condition for a group or an aggregate. The HAVING clause is often used with the GROUP BY clause to filter groups or aggregates based on a specified condition.
The `WHERE` clause allows you to filter rows based on a specified condition. However, the `HAVING` clause allows you to filter groups of rows according to a specified condition.

SELECT customer_id, SUM (amount) FROM payment GROUP BY customer_id HAVING SUM (amount) > 200;

//The `UNION` operator combines result sets of two or more `SELECT` statements into a single result set.

SELECT * FROM top_rated_films UNION SELECT * FROM most_popular_films;

SELECT select_list FROM A INTERSECT SELECT select_list FROM B;

SELECT select_list FROM A EXCEPT SELECT select_list FROM B;

//grouping set ← element를 (brand,segment) 꼴로 생각해도됨
SELECT brand, segment, SUM (quantity) FROM sales GROUP BY brand, segment;

//**Suppose that you want to get all the grouping sets by using a single query.** To achieve this, you may use the `UNION ALL` to combine all the queries above.

Because `UNION ALL` requires all result sets to have the same number of columns with compatible data types, you need to adjust the queries by adding `NULL` to the selection list of each as shown below:

SELECT brand, segment, SUM (quantity) FROM sales GROUP BY brand, segment

UNION ALL

SELECT brand, NULL, SUM (quantity) FROM sales GROUP BY brand

UNION ALL

SELECT NULL, segment, SUM (quantity) FROM sales GROUP BY segment

UNION ALL

SELECT NULL, NULL, SUM (quantity) FROM sales;

//To make it more efficient, PostgreSQL provides the `GROUPING SETS` clause which is the sub clause of the `GROUP BY` clause.

The `GROUPING SETS` allows you to define multiple grouping sets in the same query.

The general syntax of the `GROUPING SETS` is as follows:

```
SELECT c1,  c2, aggregate_function(c3) FROM  table_name
GROUP BY GROUPING SETS (
        (c1, c2),
        (c1),
        (c2),
        ()
);
```

//The `GROUPING()` function returns bit 0 if the argument is a member of the current grouping set and 1 otherwise.

```
SELECT
        GROUPING(brand) grouping_brand,
        GROUPING(segment) grouping_segment,
        brand,
        segment,
        SUM (quantity)
FROM
        sales
GROUP BY
        GROUPING SETS (
                (brand),
                (segment),
                ()
        )
ORDER BY
        brand,
        segment;
```

grouping_brand	grouping_segment	brand	segment	sum
0	1	ABC		300
0	1	XYZ		400
1	0		Basic	500
1	0		Premium	200
1	1			700

(5개 행)

//As shown in the screenshot, when the value in the `grouping_brand` is 0, the `sum` column shows the subtotal of the `brand`.

When the value in the `grouping_segment` is zero, the sum column shows the subtotal of the `segment`.

You can use the GROUPING() function in the HAVING clause to find the subtotal of each brand like this:

```
SELECT
        GROUPING(brand) grouping_brand,
        GROUPING(segment) grouping_segment,
        brand,
        segment,
        SUM (quantity)
FROM
        sales
GROUP BY
        GROUPING SETS (
                (brand),
                (segment),
                ()
        )
HAVING GROUPING(brand) = 0
ORDER BY
        brand,
        segment;
```

	grouping_brand integer	grouping_segment integer	brand character varying	segment character varying	sum bigint
1	0	1	ABC	[null]	300
2	0	1	XYZ	[null]	400

//PostgreSQL CUBE is a subclause of the GROUP BY clause. The CUBE allows you to generate multiple grouping sets.

SELECT c1,c2,c3, aggregate (c4) FROM table_name
GROUP BY
CUBE (c1, c2, c3);
The query generates all possible grouping sets based on the dimension columns specified in CUBE. The CUBE subclause is a short way to define multiple grouping sets so the following are equivalent:
GROUPING SETS ((c1,c2,c3), (c1,c2), (c1,c3), (c2,c3), (c1), (c2), (c3), ())
you will have 2^n combinations

//ROLLUP(c1,c2,c3) generates only four grouping sets, assuming the hierarchy c1 > c2 > c3 as follows:
(c1, c2, c3)
(c1, c2)
(c1)
()

Section 7. Subquery

SELECT film_id, title, rental_rate FROM film WHERE
rental_rate > (SELECT AVG (rental_rate) FROM film);

```
SELECT
	film_id,
	title
FROM
	film
WHERE
	film_id IN (
		SELECT
			inventory.film_id
		FROM
			rental INNER JOIN inventory
			ON inventory.inventory_id = rental.inventory_id
		WHERE
			return_date BETWEEN '2005-05-29' AND '2005-05-30'
	);
```

//A subquery can be an input of the `EXISTS` operator. If the subquery returns any row, the `EXISTS` operator returns true. If the subquery returns no row, the result of `EXISTS` operator is false.

The `EXISTS` operator only cares about the number of rows returned from the subquery, not the content of the rows, therefore, the common coding convention of `EXISTS` operator is as follows:

SELECT first_name, last_name FROM customer WHERE EXISTS (

SELECT 1 FROM payment WHERE payment.customer_id = customer.customer_id

);

//The PostgreSQL `ANY` operator compares a value to a set of values returned by a subquery.

The following example returns the maximum length of film grouped by film category:

```
SELECT
    MAX( length )
FROM
    film
INNER JOIN film_category
        USING(film_id)
GROUP BY
    category_id;
```

You can use this query as a subquery in the following statement that finds the films whose lengths are greater than or equal to the maximum length of any film category :

```
SELECT title
FROM film
WHERE length >= ANY(
    SELECT MAX( length )
    FROM film
    INNER JOIN film_category USING(film_id)
    GROUP BY  category_id );
```

//The PostgreSQL ALL operator allows you to query data by comparing a value with a list of values returned by a subquery.

To find all films whose lengths are greater than the list of the average lengths above, you use the ALL and greater than operator (>) as follows:

SELECT film_id, title, length FROM film WHERE length > ALL (SELECT ROUND(AVG (length),2) FROM film GROUP BY rating) ORDER BY length;

// A common table expression is a temporary result set which you can reference within another SQL statement. Common Table Expressions are temporary in the sense that they only exist during the execution of the query.

```sql
WITH cte_film AS (
    SELECT
        film_id,
        title,
        (CASE
            WHEN length < 30 THEN 'Short'
            WHEN length < 90 THEN 'Medium'
            ELSE 'Long'
        END) length
    FROM
        film
)
SELECT
    film_id,
    title,
    length
FROM
    cte_film
WHERE
    length = 'Long'
ORDER BY
    title;
```

//Case

```sql
select name,
        case when (monthlymaintenance > 100) then 'expensive'
        else 'cheap'
        end as cost
        from cd.facilities;
```

```sql
WITH cte_film AS (
    SELECT
        film_id,
        title,
        (CASE
            WHEN length < 30 THEN 'Short'
            WHEN length < 90 THEN 'Medium'
            ELSE 'Long'
        END) length
    FROM
        film
)
SELECT
    film_id,
    title,
    length
FROM
    cte_film
WHERE
    length = 'Long'
ORDER BY
    title;
```

```sql
WITH RECURSIVE subordinates AS (
	SELECT
		employee_id,
		manager_id,
		full_name
	FROM
		employees
	WHERE
		employee_id = 2
	UNION
		SELECT
			e.employee_id,
			e.manager_id,
			e.full_name
		FROM
			employees e
		INNER JOIN subordinates s ON s.employee_id = e.manager_id
) SELECT
	*
FROM
	subordinates;
```

UPDATE courses

SET published_date = '2020-08-01'

WHERE course_id = 3;