

Database Management Systems.

Lecture 4

The background of the slide features a series of concentric, curved lines in a light gray color, creating a sense of motion or a stylized globe. These lines are more prominent on the left and right sides of the slide.

Content:

Joining Multiple Tables

1. **Inner Join**
2. **Left Join**
3. **Right Join**
4. **Outer Join**
5. **Self Join**
6. **Cross Join**
7. **Natural Join**

JOINS

- PostgreSQL **JOIN** is used to combine columns from one or more tables based on the values of the common columns between related tables.
- The common columns are typically the primary key columns of the first table and foreign key columns of the second table.
- PostgreSQL supports **inner join**, **left join**, **right join**, **full outer join**, **cross join**, **natural join**, and a special kind of join called **self-join**.

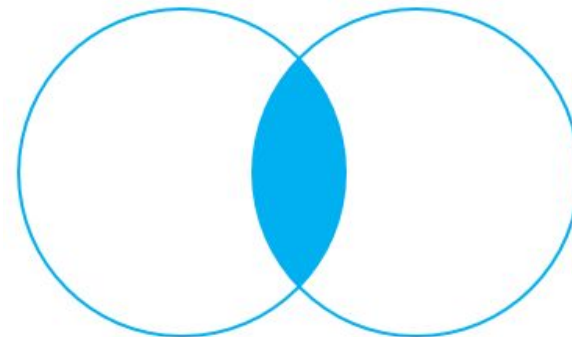
INNER JOIN

- The **INNER JOIN** keyword selects all rows from both the tables if the condition satisfies.
- This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.

- **Basic syntax:**

```
SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1 INNER JOIN table2  
ON table1.matching_column = table2.matching_column;
```

The following Venn diagram illustrates how INNER JOIN clause works:

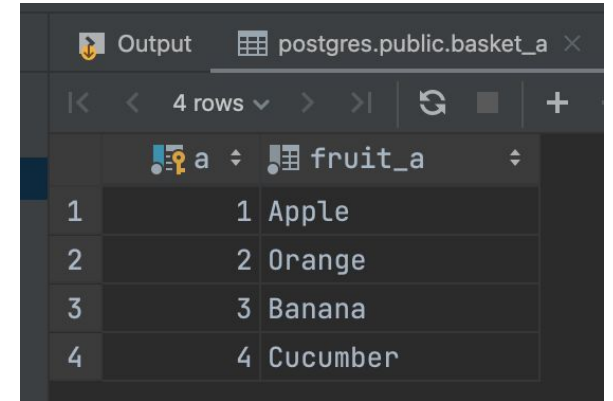


INNER JOIN

Example:

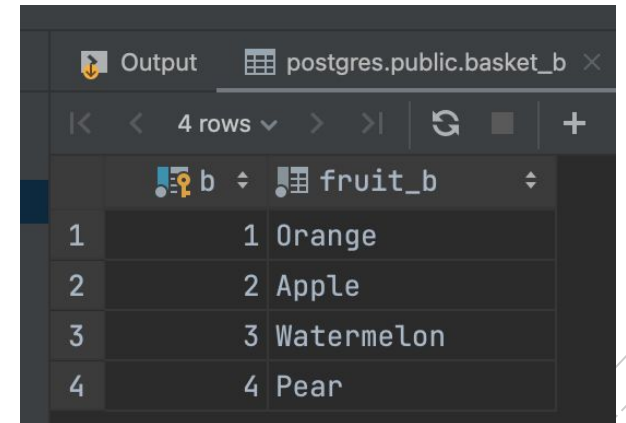
- Suppose you have two tables called `basket_a` and `basket_b` and that store fruits:

```
CREATE TABLE basket_a (  
    a INT PRIMARY KEY,  
    fruit_a VARCHAR (100) NOT NULL  
);  
  
CREATE TABLE basket_b (  
    b INT PRIMARY KEY,  
    fruit_b VARCHAR (100) NOT NULL  
);  
  
INSERT INTO basket_a (a, fruit_a)  
VALUES  
    (1, 'Apple'),  
    (2, 'Orange'),  
    (3, 'Banana'),  
    (4, 'Cucumber');  
  
INSERT INTO basket_b (b, fruit_b)  
VALUES  
    (1, 'Orange'),  
    (2, 'Apple'),  
    (3, 'Watermelon'),  
    (4, 'Pear');
```



Output postgres.public.basket_a

	a	fruit_a
1	1	Apple
2	2	Orange
3	3	Banana
4	4	Cucumber



Output postgres.public.basket_b

	b	fruit_b
1	1	Orange
2	2	Apple
3	3	Watermelon
4	4	Pear

- The tables have some common fruits such as apple and orange.

Example:

```
✓ SELECT
    a,
    fruit_a,
    b,
    fruit_b
FROM
    basket_a
INNER JOIN basket_b
    ON fruit_a = fruit_b;
```

	a	fruit_a	b	fruit_b
1	1	Apple	2	Apple
2	2	Orange	1	Orange

The inner join examines each row in the first table (basket_a). It compares the value in the fruit_a column with the value in the fruit_b column of each row in the second table (basket_b). If these values are equal, the inner join creates a new row that contains columns from both tables and adds this new row the result set.

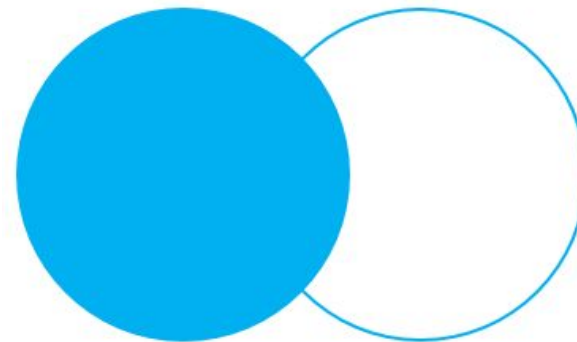
LEFT JOIN

- This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join.
- The rows for which there is no matching row on right side, the result-set will contain *null*.
- **LEFT JOIN** is also known as **LEFT OUTER JOIN**

- **Basic syntax:**

```
SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1 LEFT JOIN table2  
ON table1.matching_column = table2.matching_column;
```

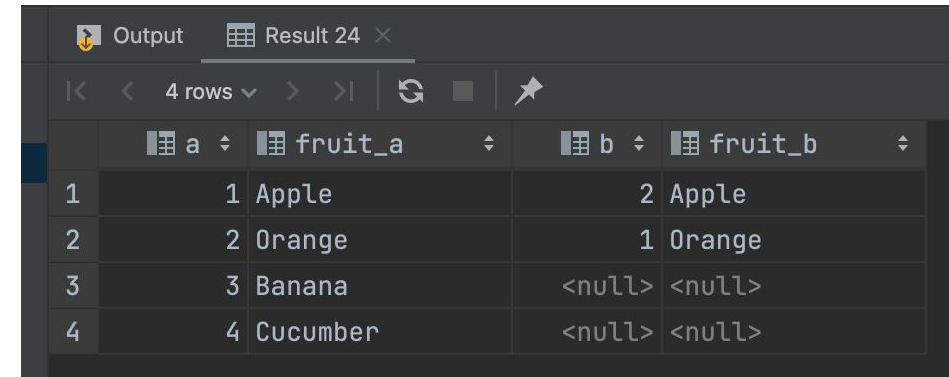
The following Venn diagram illustrates how LEFT JOIN clause works:



LEFT OUTER JOIN

Example:

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



	a	fruit_a	b	fruit_b
1	1	Apple	2	Apple
2	2	Orange	1	Orange
3	3	Banana	<null>	<null>
4	4	Cucumber	<null>	<null>

- The left join starts selecting data from the left table. It compares values in the fruit_a column with the values in the fruit_b column in the basket_b table.
- If these values are equal, the left join creates a new row that contains columns of both tables and adds this new row to the result set. (see the row #1 and #2 in the result set).
- In case the values do not equal, the left join also creates a new row that contains columns from both tables and adds it to the result set. However, it fills the columns of the right table (basket_b) with null. (see the row #3 and #4 in the result set).

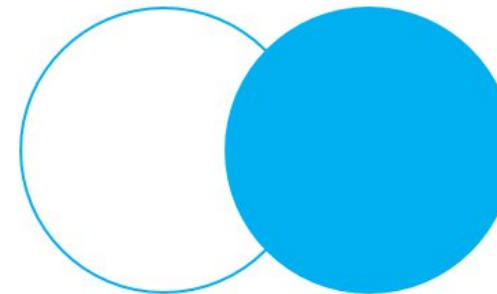
RIGHT JOIN

- **RIGHT JOIN** is similar to LEFT JOIN.
- This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join.
- The rows for which there is no matching row on left side, the result-set will contain *null*.
- **RIGHT JOIN** is also known as **RIGHT OUTER JOIN**

- **Basic syntax:**

```
SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1 RIGHT JOIN table2  
ON table1.matching_column = table2.matching_column;
```

The following Venn diagram illustrates how RIGHT JOIN clause works:



RIGHT OUTER JOIN

Example:

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

	a	fruit_a	b	fruit_b
1	2	Orange	1	Orange
2	1	Apple	2	Apple
3	<null>	<null>	3	Watermelon
4	<null>	<null>	4	Pear

- The right join is a reversed version of the left join. The right join starts selecting data from the right table. It compares each value in the fruit_b column of every row in the right table with each value in the fruit_a column of every row in the fruit_a table.
- If these values are equal, the right join creates a new row that contains columns from both tables.
- In case these values are not equal, the right join also creates a new row that contains columns from both tables. However, it fills the columns in the left table with NULL.

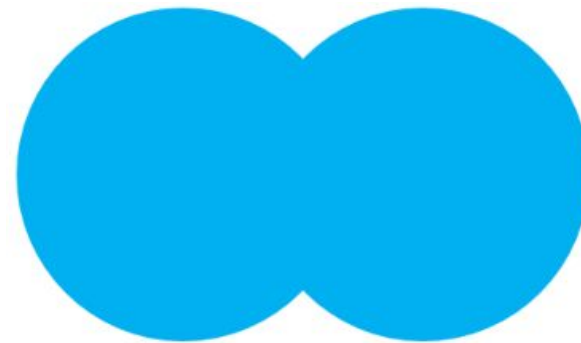
FULL JOIN

- **FULL JOIN** creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN.
- The result-set will contain all the rows from both the tables.
- The rows for which there is no matching, the result-set will contain *NULL* values

- **Basic syntax:**

```
SELECT table1.column1,table1.column2,table2.column1,....  
FROM table1 FULL JOIN table2  
ON table1.matching_column = table2.matching_column;
```

The following Venn diagram illustrates how FULL JOIN clause works:



FULL OUTER JOIN

Example:

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

	a	fruit_a	b	fruit_b
1	1	Apple	2	Apple
2	2	Orange	1	Orange
3	3	Banana	<null>	<null>
4	4	Cucumber	<null>	<null>
5	<null>	<null>	3	Watermelon
6	<null>	<null>	4	Pear

- The full outer join or full join returns a result set that contains all rows from both left and right tables, with the matching rows from both sides if available.
- In case there is no match, the columns of the table will be filled with NULL.

CROSS JOIN

- A **CROSS JOIN** clause allows you to produce a Cartesian Product of rows in two or more tables.
- Different from other join clauses such as LEFT JOIN or INNER JOIN, the CROSS JOIN clause does not have a join predicate.

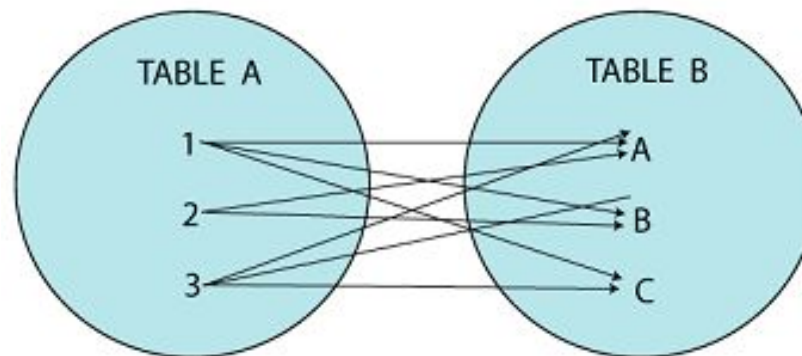
- **Basic syntax:**

```
SELECT select_list  
FROM T1 CROSS JOIN T2;
```

OR

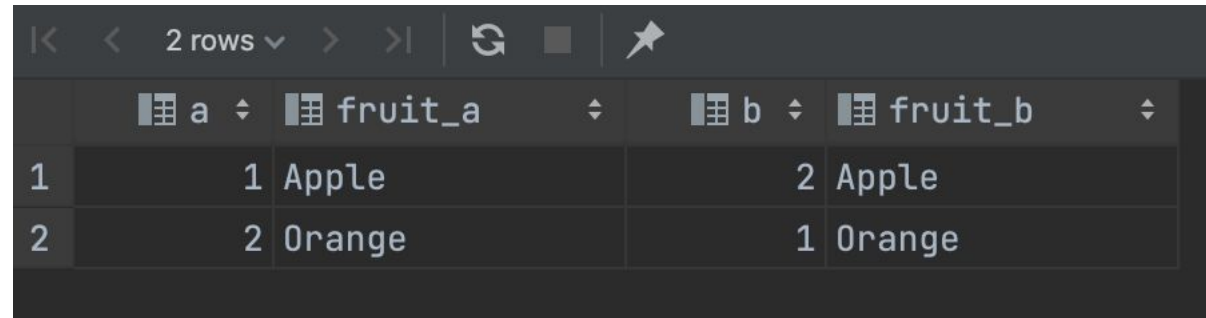
```
SELECT select_list  
FROM T1, T2;
```

CROSS JOIN



Example:

```
select * from basket_a cross join basket_b  
where basket_a.fruit_a = basket_b.fruit_b;
```



	a	fruit_a	b	fruit_b
1	1	Apple	2	Apple
2	2	Orange	1	Orange

- In this case **CROSS JOIN** works like INNER JOIN

NATURAL JOIN

equivalent to:

- A **NATURAL JOIN** is a join that creates an implicit join based on the same column names in the joined tables.
- A **NATURAL JOIN** can be an inner join or left join or right join. If you do not specify a join explicitly e.g., INNER JOIN, LEFT JOIN, RIGHT JOIN, PostgreSQL will use the INNER JOIN by default.
- If you use the asterisk (*) in the select list, the result will contain the following columns:
 - All the common columns, which are the columns from both tables that have the same name.
 - Every column from both tables, which is not a common column.

- **Basic syntax:**

```
SELECT select_list  
FROM T1 NATURAL [INNER, LEFT, RIGHT] JOIN T2;
```

```
SELECT select_list FROM T1  
INNER JOIN T2 USING (matching_column);
```

Example:

```
DROP TABLE IF EXISTS categories;
CREATE TABLE categories (
    category_id serial PRIMARY KEY,
    category_name VARCHAR (255) NOT NULL
);

DROP TABLE IF EXISTS products;
CREATE TABLE products (
    product_id serial PRIMARY KEY,
    product_name VARCHAR (255) NOT NULL,
    category_id INT NOT NULL,
    FOREIGN KEY (category_id) REFERENCES categories (category_id)
);
```

```
INSERT INTO categories (category_name)
VALUES
    ('Smart Phone'),
    ('Laptop'),
    ('Tablet');

INSERT INTO products (product_name, category_id)
VALUES
    ('iPhone', 1),
    ('Samsung Galaxy', 1),
    ('HP Elite', 2),
    ('Lenovo Thinkpad', 2),
    ('iPad', 3),
    ('Kindle Fire', 3);
```

	category_id	category_name
1	1	Smart Phone
2	2	Laptop
3	3	Tablet

	product_id	product_name	category_id
1	1	iPhone	1
2	2	Samsung Galaxy	1
3	3	HP Elite	2
4	4	Lenovo Thinkpad	2
5	5	iPad	3
6	6	Kindle Fire	3

Example:

```
select * from products natural join categories;
```

	category_id	product_id	product_name	category_name
1	1	1	iPhone	Smart Phone
2	1	2	Samsung Galaxy	Smart Phone
3	2	3	HP Elite	Laptop
4	2	4	Lenovo Thinkpad	Laptop
5	3	5	iPad	Tablet
6	3	6	Kindle Fire	Tablet

```
select * from products inner join categories  
1..n<->1: using (category_id);
```

SELF JOIN

- A **self-join** is a regular join that joins a table to itself.
- In practice, you typically use a self-join to query hierarchical data or to compare rows within the same table.
- To form a self-join, you specify the same table twice with different table aliases and provide the join predicate after the ON keyword.
- The following query uses an INNER JOIN that joins the table to itself:

```
SELECT select_list  
  
FROM table_name t1 INNER JOIN table_name t2  
ON join_predicate;
```

- Also, you can use the LEFT JOIN or RIGHT JOIN clause to join table to itself like this:

```
SELECT select_list  
  
FROM table_name t1 LEFT JOIN table_name t2  
ON join_predicate;
```

Example:

```
SELECT
  f1.title,
  f2.title,
  f1.length
FROM
  film f1
INNER JOIN film f2
  ON f1.film_id <> f2.film_id AND
     f1.length = f2.length;
```

Output			
dvdrental.public.film			
	f1.title	f2.title	length
1	Chamber Italian	Resurrection Silverado	117
2	Chamber Italian	Magic Mallrats	117
3	Chamber Italian	Graffiti Love	117
4	Chamber Italian	Affair Prejudice	117
5	Grosse Wonderful	Hurricane Affair	49

UPDATE JOIN

- Sometimes, you need to update data in a table based on values in another table. In this case, you can use the PostgreSQL **UPDATE join** syntax as follows:

```
UPDATE t1
SET t1.c1 = new_value
FROM t2
WHERE t1.c2 = t2.c2;
```

- To join to another table in the **UPDATE** statement, you specify the joined table in the **FROM** clause and provide the join condition in the **WHERE** clause. The **FROM** clause must appear immediately after the **SET** clause.
- For each row of table t1, the UPDATE statement examines every row of table t2.
- If the value in the c2 column of table t1 equals the value in the c2 column of table t2, the UPDATE statement updates the value in the c1 column of the table t1 the new value (new_value).

Example:

```
185 ✓ select * from stark;
186
187
188
```

Output postgres.public.stark

	stark_id	f_name	l_name	gender	birth_of_date	personality
1	1	Jon	Snow	male	1530-01-01	positive
2	2	Arya	Stark	female	1540-03-03	positive
3	3	Sansa	Stark	female	1532-06-05	positive

```
222
223 INSERT INTO lannister(f_name, l_name, gender, birth_of_date, personality)
224 SELECT f_name, l_name, gender, birth_of_date, personality
225 FROM stark
226 WHERE f_name = 'Sansa';
227 ✓ select * from lannister;
228
```

Output postgres.public.lannister

	lannister_id	f_name	l_name	gender	birth_of_date	personality
1	1	Cersei	Lannister	female	1512-09-30	villain
2	2	Jayne	Lannister	male	1512-09-30	villain
3	3	Tyrion	Lannister	female	1515-02-05	positive
4	4	Sansa	Stark	female	1532-06-05	positive

```
295 ✓ UPDATE stark
296 SET l_name = lannister.l_name
297 FROM lannister
298 WHERE lannister.f_name = stark.f_name
299 returning *;
300
```

Output Result 145

	stark_id	f_name	l_name	gender	birth_of_date	personality	lannister_id	f_name
1	3	Sansa	Stark	female	1532-06-05	positive	4	Sansa

DELETE JOIN

- *PostgreSQL doesn't support the **DELETE JOIN** statement. However, it does support the **USING** clause in the **DELETE** statement that provides similar functionality as the **DELETE JOIN**.*
- The following shows the syntax of the **DELETE** statement with the **USING** clause
- In this syntax:
- First, specify the table expression after the **USING** keyword. It can be one or more tables.
- Then, use columns from the tables that appear in the **USING** clause in the **WHERE** clause for joining data.
- For example, the following statement uses the **DELETE** statement with the **USING** clause to delete data from t1 that has the same id

```
DELETE FROM table_name1
USING table_expression
WHERE condition
RETURNING returning_columns;
```

```
DELETE FROM t1
USING t2
WHERE t1.id = t2.id
```

Example:

```
321 |  
322 ✓ DELETE FROM lannister  
323 USING stark  
324 WHERE lannister.f_name = stark.f_name  
325 returning *;  
326  
327
```

Output Result 146

	lannister_id	f_name	l_name	gender	birth_of_date	personality	stark_id	f_name
1		4 Sansa	Stark	female	1532-06-05	positive	3	Sansa

SEQUENCE

- By definition, a **sequence** is an ordered list of integers. The ***orders of numbers in the sequence are important***. For example, {1,2,3,4,5} and {5,4,3,2,1} are *entirely different sequences*.
- A **sequence** in PostgreSQL is a *user-defined schema-bound object that generates a sequence of integers based on a specified specification*.
- To create a sequence in PostgreSQL, you use the **CREATE SEQUENCE** statement.

SEQUENCE

- By definition, a **sequence** is an ordered list of integers. The **orders of numbers in the sequence are important**. For example, {1,2,3,4,5} and {5,4,3,2,1} are *entirely different sequences*.
- A **sequence** in PostgreSQL is a *user-defined schema-bound object that generates a sequence of integers based on a specified specification*.
- To create a sequence in PostgreSQL, you use the **CREATE SEQUENCE** statement.
- The following illustrates the syntax of the **CREATE SEQUENCE** statement

```
CREATE SEQUENCE [ IF NOT EXISTS ] sequence_name
    [ AS { SMALLINT | INT | BIGINT } ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ]
    [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ CACHE cache ]
    [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

```
CREATE SEQUENCE [ IF NOT EXISTS ] sequence_name
[ AS { SMALLINT | INT | BIGINT } ]
[ INCREMENT BY ] increment
[ MINVALUE minvalue | NO MINVALUE ]
[ MAXVALUE maxvalue | NO MAXVALUE ]
[ START [ WITH ] start ]
[ CACHE cache ]
[ [ NO ] CYCLE ]
[ OWNED BY { table_name.column_name | NONE } ]
```

- The **START** clause specifies the starting value of the sequence. The default starting value is minvalue for ascending sequences and maxvalue for descending ones.
- The **CACHE** determines how many sequence numbers are preallocated and stored in memory for faster access. One value can be generated at a time. By default, the sequence generates one value at a time i.e., no cache.
- The **CYCLE** allows you to restart the value if the limit is reached. If you use **NO CYCLE**, when the limit is reached, attempting to get the next value will result in an error. The **NO CYCLE** is the *default* if you don't explicitly specify **CYCLE** or **NO CYCLE**.
- The **OWNED BY** clause allows you to associate the table column with the sequence so that when you drop the column or table, PostgreSQL will automatically drop the associated sequence.
- **Note that when you use the SERIAL pseudo-type for a column of a table, behind the scenes, PostgreSQL automatically creates a**

- Specify the **name of the sequence** after the CREATE SEQUENCE clause.
- The **sequence name** must be distinct from any other sequences, tables, indexes, views, or foreign tables in the same schema.
- The **IF NOT EXISTS** conditionally creates a new sequence only if it does not exist.
- Specify the **data type** of the sequence. The valid data type is SMALLINT, INT, and BIGINT. The default data type is BIGINT if you skip it.
- The **increment** specifies which value to be added to the current sequence value to create new value.
- ~~A positive number will make an ascending sequence while a negative number will form a descending sequence.~~
- **The default increment value is 1.**
- Define the **minimum value** and **maximum value** of the sequence. If you use **NO MINVALUE** and **NO MAXVALUE**, the sequence will use the default value.
- For an ascending sequence, the default maximum value is the maximum value of the data type of the sequence and the default minimum value is 1.
- In case of a descending sequence, the default maximum value is -1

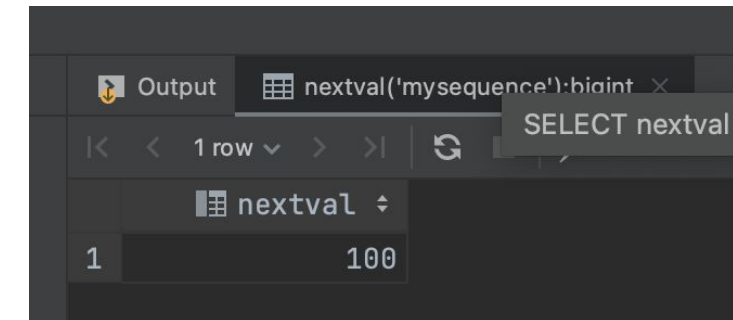
Example:

This statement uses the CREATE SEQUENCE statement to create a new ascending sequence starting from 100 with an increment of 5:

```
327  
328 ✓ CREATE SEQUENCE mysequence  
329 INCREMENT 5  
330 START 100;  
331
```

To get the next value from the sequence to you use the nextval() function:

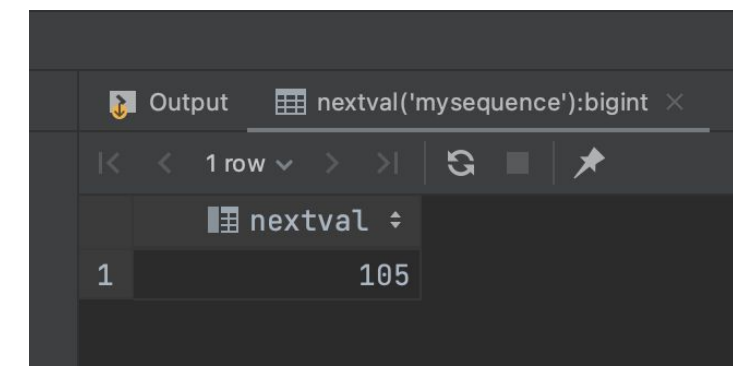
```
332  
333 ✓ SELECT nextval('mysequence');  
334
```



nextval('mysequence'):bigint	
1	100

If you execute the statement again, you will get the next value from the sequence:

```
332  
333 ✓ SELECT nextval('mysequence');  
334
```



nextval('mysequence'):bigint	
1	105

To remove the sequence from database:

```
335 ✓ drop sequence mysequence;
```