netlink
digital solutions

DATABASE

www.netlink.com

PostgreSQL SERIAL To Create Auto-increment Column

Introduction to the PostgreSQL SERIAL pseudo-type

- In PostgreSQL, a sequence is a special kind of database object that generates a sequence of integers.
- A sequence is often used as the primary key column in a table.

When creating a new table, the sequence can be created through the SERIAL pseudo-type as follows:

```
CREATE TABLE table_name(
    id SERIAL
 );
```

By assigning the SERIAL pseudo-type to the id column, PostgreSQL performs the following:

1. First, create a sequence object and set the next value generated by the sequence as the default value for the column.
2. Second, add a NOT NULL constraint to the id column because a sequence always generates an integer, which is a non-null value.
3. Third, assign the owner of the sequence to the id column; as a result, the sequence object is deleted when the id column or table is dropped.

PostgreSQL SERIAL To Create Auto-increment Column

Behind the scenes, the following statement:

```
CREATE TABLE table_name(
   id SERIAL
);
```

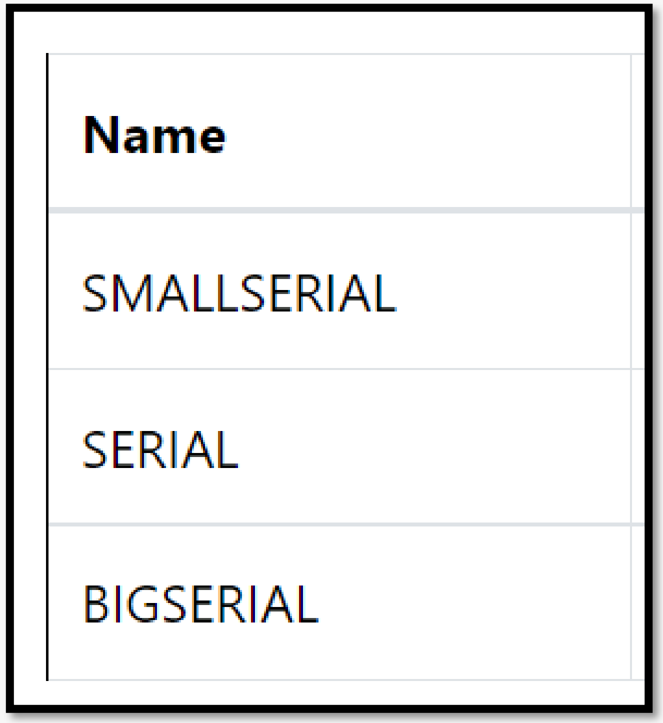
PostgreSQL provides three serial pseudo-types SMALLSERIAL, SERIAL, and BIGSERIAL with the following characteristics:

is equivalent to the following statements:

-
  ```
  CREATE SEQUENCE table_name_id_seq;
  ```

-
  ```
  CREATE TABLE table_name (
    id integer NOT NULL DEFAULT nextval('table_name_id_seq')
  );
  ```

-
  ```
  ALTER SEQUENCE table_name_id_seq OWNED BY
  table_name.id;
  ```



Name
SMALLSERIAL
SERIAL
BIGSERIAL

PostgreSQL SERIAL To Create Auto-increment Column

If you want to get the value generated by the sequence when you insert a new row into the table, you use the RETURNING id clause in the INSERT statement.

The following statement inserts a new row into the fruits table and returns the value generated for the id column.

```
INSERT INTO fruits(name)
VALUES('Banana')
RETURNING id;
```

The sequence generator operation is not transaction-safe. It means that if two concurrent database connections attempt to get the next value from a sequence, each client will get a different value. If one client rolls back the transaction, the sequence number of that client will be unused, creating a gap in the sequence.

PostgreSQL Sequences

- ✓ 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.

Introduction to PostgreSQL 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 } ]
```

PostgreSQL Sequences

PostgreSQL CREATE SEQUENCE examples

1) Creating an ascending sequence example

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

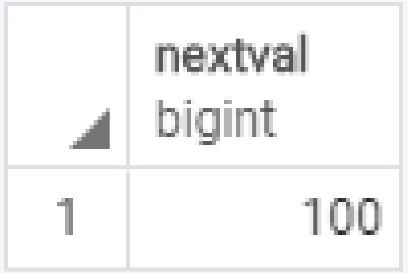
```
CREATE SEQUENCE mysequence
INCREMENT 5
START 100;
```

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

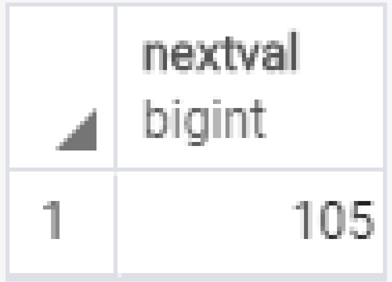
```
SELECT nextval('mysequence');
```

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

```
SELECT nextval('mysequence');
```



	nextval bigint
1	100



	nextval bigint
1	105

PostgreSQL Sequences

PostgreSQL CREATE SEQUENCE examples

2) Creating a descending sequence example

The following statement creates a descending sequence from 3 to 1 with the cycle option:

```
CREATE SEQUENCE three
INCREMENT -1
MINVALUE 1
MAXVALUE 3
START 3
CYCLE;
```

When you execute the following statement multiple times, you will see the number starting from 3, 2, 1 and back to 3, 2, 1 and so on:

```
SELECT nextval('three');
```

PostgreSQL Sequences

PostgreSQL CREATE SEQUENCE examples

3) Creating a sequence associated with a table column

First, create a new table named order_details:

```
CREATE TABLE order_details(
order_id SERIAL,
item_id INT NOT NULL,
 item_text VARCHAR NOT NULL,
price DEC(10,2) NOT NULL,
 PRIMARY KEY(order_id, item_id) );
```

Second, create a new sequence associated with the item_id column of the order_details table:

```
CREATE SEQUENCE order_item_id
START 10
INCREMENT 10
MINVALUE 10
OWNED BY order_details.item_id;
```

Third, insert three order line items into the order_details table:

```
INSERT INTO order_details(order_id, item_id, item_text,
price)
VALUES (100, nextval('order_item_id'),'DVD Player',100),
(100, nextval('order_item_id'),'Android TV',550),
(100, nextval('order_item_id'),'Speaker',250);
```

In this statement, we used the nextval() function to fetch item id value from the order_item_id sequence.

Fourth, query data from the order_details table:

```
SELECT order_id, item_id, item_text, price FROM
order_details;
```

PostgreSQL Sequences

Deleting sequences :

If a sequence is associated with a table column, it will be automatically dropped once the table column is removed or the table is dropped.

You can also remove a sequence manually using the DROP SEQUENCE statement:

```
DROP SEQUENCE [ IF EXISTS ] sequence_name [, ...]
[ CASCADE | RESTRICT ];
```

PostgreSQL Identity Column

Introduction to PostgreSQL identity column:

PostgreSQL version 10 introduced a new constraint **GENERATED AS IDENTITY** that allows you to automatically assign a unique number to a column.

The GENERATED AS IDENTITY constraint is the SQL standard-conforming variant of the good old SERIAL column.

The following illustrates the syntax of the GENERATED AS IDENTITY constraint:

column_name type GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY[(sequence_option)]

In this syntax:

1. The type can be SMALLINT, INT, or BIGINT.
2. The GENERATED ALWAYS instructs PostgreSQL to always generate a value for the identity column. If you attempt to insert (or update) values into the GENERATED ALWAYS AS IDENTITY column, PostgreSQL will issue an error.
3. The GENERATED BY DEFAULT also instructs PostgreSQL to generate a value for the identity column. However, if you supply a value for insert or update, PostgreSQL will use that value to insert into the identity column instead of using the system-generated value.

netlink | digital solutions

PostgreSQL Identity Column

PostgreSQL allows you a table to have more than one identity column. Like the **SERIAL**, the **GENERATED AS IDENTITY** constraint also uses the SEQUENCE object internally.

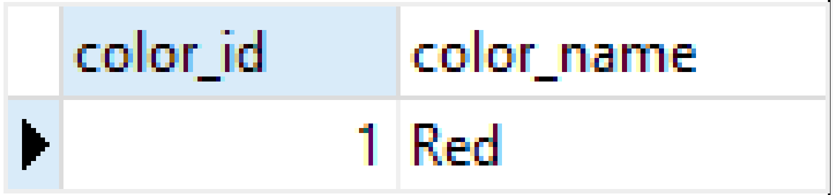
A) GENERATED ALWAYS example

First, create a table named color with the color_id as the identity column:

```
CREATE TABLE color (
color_id INT GENERATED ALWAYS AS IDENTITY,
color_name VARCHAR NOT NULL );
```

Second, insert a new row into the color table:

```
INSERT INTO color(color_name) VALUES ('Red');
```



color_id	color_name
1	Red

Because color_id column has the GENERATED AS IDENTITY constraint, PostgreSQL generates a value for it as shown in the query below:

```
SELECT * FROM color;
```

PostgreSQL Identity Column

A) GENERATED ALWAYS example

First, create a table named color with the color_id as the identity column:

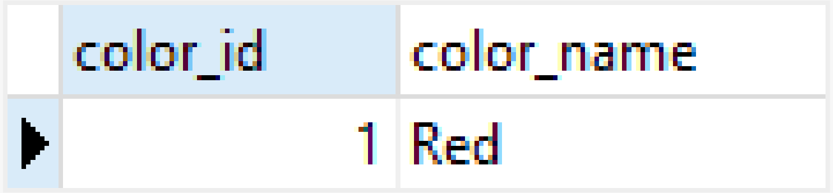
```
CREATE TABLE color (
color_id INT GENERATED ALWAYS AS IDENTITY,
color_name VARCHAR NOT NULL );
```

Second, insert a new row into the color table:

```
INSERT INTO color(color_name) VALUES ('Red');
```

Because color_id column has the GENERATED AS IDENTITY constraint, PostgreSQL generates a value for it as shown in the query below:

```
SELECT * FROM color;
```



	color_id	color_name
▶	1	Red

PostgreSQL Identity Column

A) GENERATED ALWAYS example

Third, insert a new row by supplying values for both color_id and color_name columns:

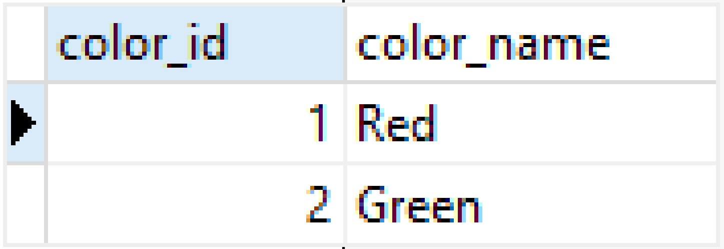
```
INSERT INTO color (color_id, color_name)
VALUES (2, 'Green');
```

PostgreSQL issued the following error:

```
[Err] ERROR:  cannot insert into column "color_id"
DETAIL:  Column "color_id" is an identity column defined as GENERATED ALWAYS.
HINT:  Use OVERRIDING SYSTEM VALUE to override.
```

To fix the error, you can use the OVERRIDING SYSTEM VALUE clause as follows:

```
INSERT INTO color (color_id, color_name)
OVERRIDING SYSTEM VALUE
VALUES(2, 'Green');
```



color_id	color_name
1	Red
2	Green

Or use GENERATED BY DEFAULT AS IDENTITY instead.

PostgreSQL Identity Column

B) GENERATED BY DEFAULT AS IDENTITY example

First, drop the color table and recreate it.

This time we use the **GENERATED BY DEFAULT AS IDENTITY** instead:

```
DROP TABLE color;

CREATE TABLE color (
color_id INT GENERATED BY DEFAULT AS IDENTITY,
color_name VARCHAR NOT NULL );
```

Second, insert a row into the color table:

```
INSERT INTO color (color_name) VALUES ('White');
```

Third, insert another row with a value for the color_id column:

```
INSERT INTO color (color_id, color_name) VALUES (2, 'Yellow');
```

Unlike the previous example that uses the GENERATED ALWAYS AS IDENTITY constraint, the statement above works perfectly fine.

PostgreSQL Identity Column

C) Sequence options example

Because the GENERATED AS IDENTITY constraint uses the SEQUENCE object, you can specify the sequence options for the system-generated values.

For example, you can specify the starting value and the increment as follows:

```
CREATE TABLE color (
color_id INT GENERATED BY DEFAULT AS IDENTITY
(START WITH 10 INCREMENT BY 10),
 color_name VARCHAR NOT NULL
);

INSERT INTO color (color_name) VALUES ('Orange');
INSERT INTO color (color_name) VALUES ('Purple');

SELECT * FROM color;
```

PostgreSQL Identity Column

Adding an identity column to an existing table

You can add identity columns to an existing table by using the following form of the ALTER TABLE statement:

```
ALTER TABLE table_name
ALTER COLUMN column_name
ADD GENERATED { ALWAYS | BY DEFAULT
} AS IDENTITY { ( sequence_option ) }
```

First, create a new table named shape:

```
CREATE TABLE shape (
shape_id INT NOT NULL,
shape_name VARCHAR NOT NULL );
```

Second, change the shape_id column to the identity column:

```
ALTER TABLE shape ALTER COLUMN shape_id ADD
GENERATED ALWAYS AS IDENTITY;
```

Note that the shape_id needs to have the NOT NULL constraint so that it can be changed to an identity column. Otherwise, you'll get an error as follows:

```
ERROR: column "shape_id" of relation "shape" must be declared NOT
NULL before identity can be added SQL state: 55000
```

PostgreSQL Identity Column

Changing an identity column

You can change the characteristics of an existing identity column by using the following ALTER TABLE statement:

```
ALTER TABLE table_name
ALTER COLUMN column_name
{ SET GENERATED { ALWAYS| BY DEFAULT } |
  SET sequence_option | RESTART [ [ WITH ]
restart ] }
```

For example, the following statement changes the shape_id column of the shape table to **GENERATED BY DEFAULT:**

```
ALTER TABLE shape
ALTER COLUMN shape_id SET GENERATED BY
DEFAULT;
```

PostgreSQL Identity Column

Removing the GENERATED AS IDENTITY constraint

The following statement removes the GENERATED AS IDENTITY constraint from an existing table:

```
ALTER TABLE table_name
ALTER COLUMN column_name
DROP IDENTITY [ IF EXISTS ]
```

For example, you can remove the GENERATED AS IDENTITY constraint column from the shape_id column of the shape table as follows:

```
ALTER TABLE shape
ALTER COLUMN shape_id
DROP IDENTITY IF EXISTS;
```