NAME

CREATE_SEQUENCE − define a new sequence generator

SYNOPSIS

CREATE [{ TEMPORARY | TEMP } | UNLOGGED] SEQUENCE [IF NOT EXISTS] *name*
[AS *data_type*]
[INCREMENT [BY] *increment*]
[MINVALUE *minvalue* | NO MINVALUE] [MAXVALUE *maxvalue* | NO MAXVALUE]
[[NO] CYCLE]
[START [WITH] *start*]
[CACHE *cache*]
[OWNED BY { *table_name*.*column_name* | NONE }]

DESCRIPTION

CREATE SEQUENCE creates a new sequence number generator. This involves creating and initializing a new special single−row table with the name *name*. The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name cannot be given when creating a temporary sequence. The sequence name must be distinct from the name of any other relation (table, sequence, index, view, materialized view, or foreign table) in the same schema.

After a sequence is created, you use the functions **nextval**, **currval**, and **setval** to operate on the sequence. These functions are documented in Section 9.17.

Although you cannot update a sequence directly, you can use a query like:

SELECT * FROM *name*;

to examine the parameters and current state of a sequence. In particular, the last_value field of the sequence shows the last value allocated by any session. (Of course, this value might be obsolete by the time it's printed, if other sessions are actively doing **nextval** calls.)

PARAMETERS

TEMPORARY or TEMP
: If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema−qualified names.

UNLOGGED
: If specified, the sequence is created as an unlogged sequence. Changes to unlogged sequences are not written to the write−ahead log. They are not crash−safe: an unlogged sequence is automatically reset to its initial state after a crash or unclean shutdown. Unlogged sequences are also not replicated to standby servers.

  Unlike unlogged tables, unlogged sequences do not offer a significant performance advantage. This option is mainly intended for sequences associated with unlogged tables via identity columns or serial columns. In those cases, it usually wouldn't make sense to have the sequence WAL−logged and replicated but not its associated table.

IF NOT EXISTS
: Do not throw an error if a relation with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing relation is anything like the sequence that would have been created — it might not even be a sequence.

name
: The name (optionally schema−qualified) of the sequence to be created.

data_type

The optional clause AS *data_type* specifies the data type of the sequence. Valid types are smallint, integer, and bigint. bigint is the default. The data type determines the default minimum and maximum values of the sequence.

increment
: The optional clause INCREMENT BY *increment* specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

minvalue
NO MINVALUE
: The optional clause MINVALUE *minvalue* determines the minimum value a sequence can generate. If this clause is not supplied or **NO MINVALUE** is specified, then defaults will be used. The default for an ascending sequence is 1. The default for a descending sequence is the minimum value of the data type.

maxvalue
NO MAXVALUE
: The optional clause MAXVALUE *maxvalue* determines the maximum value for the sequence. If this clause is not supplied or **NO MAXVALUE** is specified, then default values will be used. The default for an ascending sequence is the maximum value of the data type. The default for a descending sequence is −1.

CYCLE
NO CYCLE
: The CYCLE option allows the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively.

  If NO CYCLE is specified, any calls to **nextval** after the sequence has reached its maximum value will return an error. If neither CYCLE or NO CYCLE are specified, NO CYCLE is the default.

start
: The optional clause START WITH *start* allows the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache
: The optional clause CACHE *cache* specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache), and this is also the default.

OWNED BY *table_name.column_name*
OWNED BY NONE
: The OWNED BY option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. The specified table must have the same owner and be in the same schema as the sequence. OWNED BY NONE, the default, specifies that there is no such association.

NOTES

Use **DROP SEQUENCE** to remove a sequence.

Sequences are based on bigint arithmetic, so the range cannot exceed the range of an eight−byte integer (−9223372036854775808 to 9223372036854775807).

Because **nextval** and **setval** calls are never rolled back, sequence objects cannot be used if “gapless” assignment of sequence numbers is needed. It is possible to build gapless assignment by using exclusive locking of a table containing a counter; but this solution is much more expensive than sequence objects, especially if many transactions need sequence numbers concurrently.

Unexpected results might be obtained if a *cache* setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Each session will allocate and cache successive sequence

values during one access to the sequence object and increase the sequence object's last_value accordingly. Then, the next *cache*−1 uses of **nextval** within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in “holes” in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values might be generated out of sequence when all the sessions are considered. For example, with a *cache* setting of 10, session A might reserve values 1..10 and return **nextval**=1, then session B might reserve values 11..20 and return **nextval**=11 before session A has generated **nextval**=2. Thus, with a *cache* setting of one it is safe to assume that **nextval** values are generated sequentially; with a *cache* setting greater than one you should only assume that the **nextval** values are all distinct, not that they are generated purely sequentially. Also, last_value will reflect the latest value reserved by any session, whether or not it has yet been returned by **nextval**.

Another consideration is that a **setval** executed on such a sequence will not be noticed by other sessions until they have used up any preallocated values they have cached.

EXAMPLES

Create an ascending sequence called serial, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence:

```
SELECT nextval('serial');

 nextval
---------
     101
```

Select the next number from this sequence:

```
SELECT nextval('serial');

 nextval
---------
     102
```

Use this sequence in an **INSERT** command:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Update the sequence value after a **COPY FROM**:

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

COMPATIBILITY

CREATE SEQUENCE conforms to the SQL standard, with the following exceptions:

- Obtaining the next value is done using the **nextval()** function instead of the standard's **NEXT VALUE FOR** expression.
- The OWNED BY clause is a PostgreSQL extension.

SEE ALSO

ALTER SEQUENCE (**ALTER_SEQUENCE**(7)), DROP SEQUENCE (**DROP_SEQUENCE**(7))