Unknown Title

⋮ 13/11/2025

CREATE TABLE

CREATE TABLE — define a new table

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
EXISTS ]
 table_name
 ( [
  {
 column_name

 data_type
 [ STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT } ] [ COMPRESSION
 compression_method
 ] [ COLLATE
 collation
 ] [
 column_constraint
 [ ... ] ]
    |
 table_constraint

    | LIKE
 source_table
 [
 like_option
 ... ] }
```

```
    [, ... ]
] )
[ INHERITS (
 parent_table
 [, ... ] ) ]
[ PARTITION BY { RANGE | LIST | HASH } ( {
 column_name
 | (
 expression
 ) } [ COLLATE
 collation
 ] [
 opclass
 ] [, ... ] ) ]
[ USING
 method
 ]
[ WITH (
 storage_parameter
 [=
 value
] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE
 tablespace_name
 ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
EXISTS ]
 table_name

    OF
 type_name
 [ (
  {
 column_name
 [ WITH OPTIONS ] [
 column_constraint
 [ ... ] ]
```

```
    |
 table_constraint
 }
    [, ... ]
) ]
[ PARTITION BY { RANGE | LIST | HASH } ( {
 column_name
 | (
 expression
 ) } [ COLLATE
 collation
 ] [
 opclass
 ] [, ... ] ) ]
[ USING
 method
 ]
[ WITH (
 storage_parameter
 [=
 value
] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE
 tablespace_name
 ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
EXISTS ]
 table_name

    PARTITION OF
 parent_table
 [ (
  {
 column_name
 [ WITH OPTIONS ] [
 column_constraint
 [ ... ] ]
```

```
    |
table_constraint
 }
    [, ... ]
) ] { FOR VALUES
partition_bound_spec
 | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( {
column_name
 | (
expression
 ) } [ COLLATE
collation
 ] [
opclass
 ] [, ... ] ) ]
[ USING
method
 ]
[ WITH (
storage_parameter
 [=
value
] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE
tablespace_name
 ]

where
column_constraint
 is:

[ CONSTRAINT
constraint_name
 ]
{ NOT NULL [ NO INHERIT ]  |
  NULL |
  CHECK (
```

```
expression
```

```
) [ NO INHERIT ] |
  DEFAULT
```

```
default_expr
```

```
|
  GENERATED ALWAYS AS (
```

```
generation_expr
```

```
) [ STORED | VIRTUAL ] |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ (
```

```
sequence_options
```

```
) ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ]
```

```
index_parameters
```

```
|
  PRIMARY KEY
```

```
index_parameters
```

```
|
  REFERENCES
```

```
reftable
```

```
[ (
```

```
refcolumn
```

```
) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE
```

```
referential_action
```

```
] [ ON UPDATE
```

```
referential_action
```

```
] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
[ ENFORCED | NOT ENFORCED ]
```

and

```
table_constraint
```

```
is:

[ CONSTRAINT
```

```
constraint_name
```

```
]
```

```
{ CHECK (
```
expression
```
 ) [ NO INHERIT ] |
  NOT NULL
```
column_name
```
 [ NO INHERIT ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] (
```
column_name
```
 [, ... ] [,
```
column_name
```
 WITHOUT OVERLAPS ] )
```
index_parameters
```
 |
  PRIMARY KEY (
```
column_name
```
 [, ... ] [,
```
column_name
```
 WITHOUT OVERLAPS ] )
```
index_parameters
```
 |
  EXCLUDE [ USING
```
index_method
```
 ] (
```
exclude_element
```
 WITH
```
operator
```
 [, ... ] )
```
index_parameters
```
 [ WHERE (
```
predicate
```
 ) ] |
  FOREIGN KEY (
```
column_name
```
 [, ... ] [, PERIOD
```
column_name
```
 ] ) REFERENCES
```
reftable

```
 [ (
 refcolumn
 [, ... ] [, PERIOD
 refcolumn
 ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE
 referential_action
 ] [ ON UPDATE
 referential_action
 ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
[ ENFORCED | NOT ENFORCED ]

and
 like_option
 is:

{ INCLUDING | EXCLUDING } { COMMENTS | COMPRESSION | CONSTRAINTS | DEFAULTS |
GENERATED | IDENTITY | INDEXES | STATISTICS | STORAGE | ALL }

and
 partition_bound_spec
 is:

IN (
 partition_bound_expr
 [, ...] ) |
FROM ( {
 partition_bound_expr
 | MINVALUE | MAXVALUE } [, ...] )
  TO ( {
 partition_bound_expr
 | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS
 numeric_literal
, REMAINDER
 numeric_literal
 )
```

index_parameters

in

UNIQUE

,

PRIMARY KEY

, and

EXCLUDE

constraints are:

[INCLUDE (

column_name

[, ...])]
[WITH (

storage_parameter

[=

value

] [, ...])]
[USING INDEX TABLESPACE

tablespace_name

]

exclude_element

in an

EXCLUDE

constraint is:

{

column_name

| (

expression

) } [COLLATE

collation

] [

opclass

[(

opclass_parameter

```
 =
 value
 [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]

 referential_action
 in a
 FOREIGN KEY
/
 REFERENCES
 constraint is:

{ NO ACTION | RESTRICT | CASCADE | SET NULL [ (
 column_name
 [, ... ] ) ] | SET DEFAULT [ (
 column_name
 [, ... ] ) ] }
```

Description

`CREATE TABLE` will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

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

`CREATE TABLE` also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

The optional constraint clauses specify constraints (tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience for use when the constraint only affects one column.

To be able to create a table, you must have `USAGE` privilege on all column types or the type in the `OF` clause, respectively.

Parameters

`TEMPORARY` or `TEMP` #

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see `ON COMMIT` below). The default search_path includes the temporary schema first and so identically named existing permanent tables are not chosen for new plans while the temporary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

The autovacuum daemon cannot access and therefore cannot vacuum or analyze temporary tables. For this reason, appropriate vacuum and analyze operations should be performed via session SQL commands. For example, if a temporary table is going to be used in complex queries, it is wise to run `ANALYZE` on the temporary table after it is populated.

Optionally, `GLOBAL` or `LOCAL` can be written before `TEMPORARY` or `TEMP`. This presently makes no difference in PostgreSQL and is deprecated; see Compatibility below.

`UNLOGGED` #

If specified, the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead log (see Chapter 28), which makes them considerably faster than ordinary tables. However, they are not crash-safe: an unlogged table is automatically truncated after a crash or unclean shutdown. The contents of an unlogged table are also not replicated to standby servers. Any indexes created on an unlogged table are automatically unlogged as well.

If this is specified, any sequences created together with the unlogged table (for identity or serial columns) are also created as unlogged.

This form is not supported for partitioned tables.

`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 one that would have been created.

`table_name` #

The name (optionally schema-qualified) of the table to be created.

`OF` *`type_name`* #

Creates a *typed table*, which takes its structure from the specified stand-alone composite type (that is, one created using CREATE TYPE) though it still produces a new composite type as well. The table will have a dependency on the referenced type, meaning that cascaded alter and drop actions on that type will propagate to the table.

A typed table always has the same column names and data types as the type it is derived from, so you cannot specify additional columns. But the `CREATE TABLE` command can add defaults and constraints to the table, as well as specify storage parameters.

`column_name` #

The name of a column to be created in the new table.

`data_type` #

The data type of the column. This can include array specifiers. For more information on the data types supported by PostgreSQL, refer to Chapter 8.

`COLLATE` *`collation`* #

The `COLLATE` clause assigns a collation to the column (which must be of a collatable data type). If not specified, the column data type's default collation is used.

`STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }` #

This form sets the storage mode for the column. This controls whether this column is held inline or in a secondary TOAST table, and whether the data should be compressed or not. `PLAIN` must be used for fixed-length values such as `integer` and is inline, uncompressed. `MAIN` is for inline, compressible data. `EXTERNAL` is for external, uncompressed data, and `EXTENDED` is for external, compressed data. Writing `DEFAULT` sets the storage mode to the default mode for the column's data type. `EXTENDED` is the default for most data types that support non-`PLAIN` storage. Use of `EXTERNAL` will make substring operations on very large `text` and `bytea` values run faster, at the penalty of increased storage space. See Section 66.2 for more information.

`COMPRESSION` *`compression_method`* #

The `COMPRESSION` clause sets the compression method for the column. Compression is supported only for variable-width data types, and is used only when the column's storage mode is `main` or `extended`. (See ALTER TABLE for information on column storage modes.) Setting this property for a partitioned table has no direct effect, because such tables have no storage of their own, but the configured value will be inherited by newly-created partitions. The supported compression methods are `pglz` and `lz4`. (`lz4` is available only if `--with-lz4` was used when building PostgreSQL.) In addition, *`compression_method`* can be `default` to explicitly specify the default behavior, which is to consult the default_toast_compression setting at the time of data insertion to determine the method to use.

`INHERITS (` *`parent_table`* `[, ... ] )` #

The optional `INHERITS` clause specifies a list of tables from which the new table automatically inherits all columns. Parent tables can be plain tables or foreign tables.

Use of `INHERITS` creates a persistent relationship between the new child table and its parent table(s). Schema modifications to the parent(s) normally propagate to children as well, and by default the data of the child table is included in scans of the parent(s).

If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match in each of the parent tables. If there is no conflict, then the duplicate columns are merged to form a single column in the new table. If the column name list of the new table contains a column name that is also inherited, the data type must likewise match the inherited column(s), and the column definitions are merged into one. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations of the column. Otherwise, any parents that specify default values for the column must all specify the same default, or an error will be reported.

`CHECK` constraints are merged in essentially the same way as columns: if multiple parent tables and/or the new table definition contain identically-named `CHECK` constraints, these constraints must all have the same check expression, or an error will be reported. Constraints having the same name and expression will be merged into one copy. A constraint marked `NO INHERIT` in a parent will not be considered. Notice that an unnamed `CHECK` constraint in the new table will never be merged, since a unique name will always be chosen for it.

Column `STORAGE` settings are also copied from parent tables.

If a column in the parent table is an identity column, that property is not inherited. A column in the child table can be declared identity column if desired.

`PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ opclass ] [, ...] )` #

The optional `PARTITION BY` clause specifies a strategy of partitioning the table. The table thus created is called a *partitioned* table. The parenthesized list of columns or expressions forms the *partition key* for the table. When using range or hash partitioning, the partition key can include multiple columns or expressions (up to 32, but this limit can be altered when building PostgreSQL), but for list partitioning, the partition key must consist of a single column or expression.

Range and list partitioning require a btree operator class, while hash partitioning requires a hash operator class. If no operator class is specified explicitly, the default operator class of the appropriate type will be used; if no default operator class exists, an error will be raised. When hash partitioning is used, the operator class used must implement support function 2 (see Section 36.16.3 for details).

A partitioned table is divided into sub-tables (called partitions), which are created using separate `CREATE TABLE` commands. The partitioned table is itself empty. A data row inserted into the table is routed to a partition based on the value of columns or expressions in the partition key. If no existing partition matches the values in the new row, an error will be reported.

See Section 5.12 for more discussion on table partitioning.

`PARTITION OF parent_table { FOR VALUES partition_bound_spec | DEFAULT }` #

Creates the table as a *partition* of the specified parent table. The table can be created either as a partition for specific values using `FOR VALUES` or as a default partition using `DEFAULT`. Any indexes, constraints and user-defined row-level triggers that exist in the parent table are cloned on the new partition.

The *partition_bound_spec* must correspond to the partitioning method and partition key of the parent table, and must not overlap with any existing partition of that parent. The form with `IN` is used for list partitioning, the form with `FROM` and `TO` is used for range partitioning, and the form with `WITH` is used for hash partitioning.

partition_bound_expr is any variable-free expression (subqueries, window functions, aggregate functions, and set-returning functions are not allowed). Its data type must match the data type of the corresponding partition key column. The expression is evaluated once at table creation time, so it can even contain volatile expressions such as `CURRENT_TIMESTAMP`.

When creating a list partition, `NULL` can be specified to signify that the partition allows the partition key column to be null. However, there cannot be more than one such list partition for a given parent table. `NULL` cannot be specified for range partitions.

When creating a range partition, the lower bound specified with `FROM` is an inclusive bound, whereas the upper bound specified with `TO` is an exclusive bound. That is, the values specified in the `FROM` list are valid values of the corresponding partition key columns for this partition, whereas those in the `TO` list are not. Note that this statement must be understood according to the rules of row-wise comparison (Section 9.25.5). For example, given `PARTITION BY RANGE (x,y)`, a partition bound `FROM (1, 2) TO (3, 4)` allows x=1 with any y>=2, x=2 with any non-null y, and x=3 with any y<4.

The special values `MINVALUE` and `MAXVALUE` may be used when creating a range partition to indicate that there is no lower or upper bound on the column's value. For example, a partition defined using `FROM (MINVALUE) TO (10)` allows any values less than 10, and a partition defined using `FROM (10) TO (MAXVALUE)` allows any values greater than or equal to 10.

When creating a range partition involving more than one column, it can also make sense to use `MAXVALUE` as part of the lower bound, and `MINVALUE` as part of the upper bound. For example, a partition defined using `FROM (0, MAXVALUE) TO (10, MAXVALUE)` allows any rows where the first partition key column is greater than 0 and less than or equal to 10. Similarly, a partition defined using `FROM ('a', MINVALUE) TO ('b', MINVALUE)` allows any rows where the first partition key column starts with "a".

Note that if `MINVALUE` or `MAXVALUE` is used for one column of a partitioning bound, the same value must be used for all subsequent columns. For example, `(10, MINVALUE, 0)` is not a valid bound; you should write `(10, MINVALUE, MINVALUE)`.

Also note that some element types, such as `timestamp`, have a notion of "infinity", which is just another value that can be stored. This is different from `MINVALUE` and `MAXVALUE`, which are not real values that can be stored, but rather they are ways of saying that the value is unbounded. `MAXVALUE` can be thought of as being greater than any other value, including "infinity" and `MINVALUE` as being less than any other value, including "minus infinity". Thus the range `FROM` ('infinity') TO (MAXVALUE) is not an empty range; it allows precisely one value to be stored — "infinity".

If DEFAULT is specified, the table will be created as the default partition of the parent table. This option is not available for hash-partitioned tables. A partition key value not fitting into any other partition of the given parent will be routed to the default partition.

When a table has an existing DEFAULT partition and a new partition is added to it, the default partition must be scanned to verify that it does not contain any rows which properly belong in the new partition. If the default partition contains a large number of rows, this may be slow. The scan will be skipped if the default partition is a foreign table or if it has a constraint which proves that it cannot contain rows which should be placed in the new partition.

When creating a hash partition, a modulus and remainder must be specified. The modulus must be a positive integer, and the remainder must be a non-negative integer less than the modulus. Typically, when initially setting up a hash-partitioned table, you should choose a modulus equal to the number of partitions and assign every table the same modulus and a different remainder (see examples, below). However, it is not required that every partition have the same modulus, only that every modulus which occurs among the partitions of a hash-partitioned table is a factor of the next larger modulus. This allows the number of partitions to be increased incrementally without needing to move all the data at once. For example, suppose you have a hash-partitioned table with 8 partitions, each of which has modulus 8, but find it necessary to increase the number of partitions to 16. You can detach one of the modulus-8 partitions, create two new modulus-16 partitions covering the same portion of the key space (one with a remainder equal to the remainder of the detached partition, and the other with a remainder equal to that value plus 8), and repopulate them with data. You can then repeat this -- perhaps at a later time -- for each modulus-8 partition until none remain. While this may still involve a large amount of data movement at each step, it is still better than having to create a whole new table and move all the data at once.

A partition must have the same column names and types as the partitioned table to which it belongs. Modifications to the column names or types of a partitioned table will automatically propagate to all partitions. CHECK constraints will be inherited automatically by every partition, but an individual partition may specify additional CHECK constraints; additional constraints with the same name and condition as in the parent will be merged with the parent constraint. Defaults may be specified separately for each partition. But note that a partition's default value is not applied when inserting a tuple through a partitioned table.

Rows inserted into a partitioned table will be automatically routed to the correct partition. If no suitable partition exists, an error will occur.

Operations such as TRUNCATE which normally affect a table and all of its inheritance children will cascade to all partitions, but may also be performed on an individual partition.

Note that creating a partition using PARTITION OF requires taking an ACCESS EXCLUSIVE lock on the parent partitioned table. Likewise, dropping a partition with DROP TABLE requires taking an ACCESS EXCLUSIVE lock on the parent table. It is possible to use ALTER TABLE

ATTACH/DETACH PARTITION to perform these operations with a weaker lock, thus reducing interference with concurrent operations on the partitioned table.

`LIKE` *`source_table`* `[` *`like_option`* `... ]` #

The `LIKE` clause specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints.

Unlike `INHERITS`, the new table and original table are completely decoupled after creation is complete. Changes to the original table will not be applied to the new table, and it is not possible to include data of the new table in scans of the original table.

Also unlike `INHERITS`, columns and constraints copied by `LIKE` are not merged with similarly named columns and constraints. If the same name is specified explicitly or in another `LIKE` clause, an error is signaled.

The optional *`like_option`* clauses specify which additional properties of the original table to copy. Specifying `INCLUDING` copies the property, specifying `EXCLUDING` omits the property. `EXCLUDING` is the default. If multiple specifications are made for the same kind of object, the last one is used. The available options are:

`INCLUDING COMMENTS` #

Comments for the copied columns, constraints, and indexes will be copied. The default behavior is to exclude comments, resulting in the copied columns and constraints in the new table having no comments.

`INCLUDING COMPRESSION` #

Compression method of the columns will be copied. The default behavior is to exclude compression methods, resulting in columns having the default compression method.

`INCLUDING CONSTRAINTS` #

`CHECK` constraints will be copied. No distinction is made between column constraints and table constraints. Not-null constraints are always copied to the new table.

`INCLUDING DEFAULTS` #

Default expressions for the copied column definitions will be copied. Otherwise, default expressions are not copied, resulting in the copied columns in the new table having null defaults. Note that copying defaults that call database-modification functions, such as `nextval`, may create a functional linkage between the original and new tables.

`INCLUDING GENERATED` #

Any generation expressions as well as the stored/virtual choice of copied column definitions will be copied. By default, new columns will be regular base columns.

`INCLUDING IDENTITY` #

Any identity specifications of copied column definitions will be copied. A new sequence is created for each identity column of the new table, separate from the sequences associated with the old table.

`INCLUDING INDEXES` #

Indexes, `PRIMARY KEY`, `UNIQUE`, and `EXCLUDE` constraints on the original table will be created on the new table. Names for the new indexes and constraints are chosen according to the default rules, regardless of how the originals were named. (This behavior avoids possible duplicate-name failures for the new indexes.)

`INCLUDING STATISTICS` #

Extended statistics are copied to the new table.

`INCLUDING STORAGE` #

`STORAGE` settings for the copied column definitions will be copied. The default behavior is to exclude `STORAGE` settings, resulting in the copied columns in the new table having type-specific default settings. For more on `STORAGE` settings, see Section 66.2.

`INCLUDING ALL` #

`INCLUDING ALL` is an abbreviated form selecting all the available individual options. (It could be useful to write individual `EXCLUDING` clauses after `INCLUDING ALL` to select all but some specific options.)

The `LIKE` clause can also be used to copy column definitions from views, foreign tables, or composite types. Inapplicable options (e.g., `INCLUDING INDEXES` from a view) are ignored.

`CONSTRAINT` *`constraint_name`* #

An optional name for a column or table constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like `col must be positive` can be used to communicate helpful constraint information to client applications. (Double-quotes are needed to specify constraint names that contain spaces.) If a constraint name is not specified, the system generates a name.

`NOT NULL [ NO INHERIT ]` #

The column is not allowed to contain null values.

A constraint marked with `NO INHERIT` will not propagate to child tables.

`NULL` #

The column is allowed to contain null values. This is the default.

This clause is only provided for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

`CHECK ( expression ) [ NO INHERIT ]` #

The `CHECK` clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or UNKNOWN succeed. Should any row of an insert or update operation produce a FALSE result, an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint can reference multiple columns.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row (see Section 5.5.1). The system column `tableoid` may be referenced, but not any other system column.

A constraint marked with `NO INHERIT` will not propagate to child tables.

When a table has multiple `CHECK` constraints, they will be tested for each row in alphabetical order by name, after checking `NOT NULL` constraints. (PostgreSQL versions before 9.5 did not honor any particular firing order for `CHECK` constraints.)

`DEFAULT default_expr` #

The `DEFAULT` clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (in particular, cross-references to other columns in the current table are not allowed). Subqueries are not allowed either. The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

`GENERATED ALWAYS AS ( generation_expr ) [ STORED | VIRTUAL ]` #

This clause creates the column as a *generated column*. The column cannot be written to, and when read the result of the specified expression will be returned.

When `VIRTUAL` is specified, the column will be computed when it is read, and it will not occupy any storage. When `STORED` is specified, the column will be computed on write and will be stored on disk. `VIRTUAL` is the default.

The generation expression can refer to other columns in the table, but not other generated columns. Any functions and operators used must be immutable. References to other tables are not allowed.

A virtual generated column cannot have a user-defined type, and the generation expression of a virtual generated column must not reference user-defined functions or types, that is, it can only use built-in functions or types. This applies also indirectly, such as for functions or types that underlie operators or casts. (This restriction does not exist for stored generated columns.)

`GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ]` #

This clause creates the column as an *identity column*. It will have an implicit sequence attached to it and in newly-inserted rows the column will automatically have values from the sequence assigned to it. Such a column is implicitly `NOT NULL`.

The clauses `ALWAYS` and `BY DEFAULT` determine how explicitly user-specified values are handled in `INSERT` and `UPDATE` commands.

In an `INSERT` command, if `ALWAYS` is selected, a user-specified value is only accepted if the `INSERT` statement specifies `OVERRIDING SYSTEM VALUE`. If `BY DEFAULT` is selected, then the user-specified value takes precedence. See INSERT for details. (In the `COPY` command, user-specified values are always used regardless of this setting.)

In an `UPDATE` command, if `ALWAYS` is selected, any update of the column to any value other than `DEFAULT` will be rejected. If `BY DEFAULT` is selected, the column can be updated normally. (There is no `OVERRIDING` clause for the `UPDATE` command.)

The optional *`sequence_options`* clause can be used to override the parameters of the sequence. The available options include those shown for CREATE SEQUENCE, plus `SEQUENCE NAME` *`name`*, `LOGGED`, and `UNLOGGED`, which allow selection of the name and persistence level of the sequence. Without `SEQUENCE NAME`, the system chooses an unused name for the sequence. Without `LOGGED` or `UNLOGGED`, the sequence will have the same persistence level as the table.

`UNIQUE [ NULLS [ NOT ] DISTINCT ]` (column constraint)
`UNIQUE [ NULLS [ NOT ] DISTINCT ] (` *`column_name`* `[, ... ] [,` *`column_name`* `WITHOUT OVERLAPS ] )` [`INCLUDE (` *`column_name`* `[, ...]`)] (table constraint) #

The `UNIQUE` constraint specifies that a group of one or more columns of a table can contain only unique values. The behavior of a unique table constraint is the same as that of a unique column constraint, with the additional capability to span multiple columns. The constraint therefore enforces that any two rows must differ in at least one of these columns.

If the `WITHOUT OVERLAPS` option is specified for the last column, then that column is checked for overlaps instead of equality. In that case, the other columns of the constraint will allow duplicates so long as the duplicates don't overlap in the `WITHOUT OVERLAPS` column. (This is sometimes called a temporal key, if the column is a range of dates or timestamps, but PostgreSQL allows ranges over any base type.) In effect, such a constraint is enforced with an `EXCLUDE` constraint rather than a `UNIQUE` constraint. So for example `UNIQUE (id, valid_at WITHOUT OVERLAPS)` behaves like `EXCLUDE USING GIST (id WITH =, valid_at WITH &&)`. The `WITHOUT OVERLAPS` column must have a range or multirange type. Empty ranges/multiranges are not permitted. The non-`WITHOUT OVERLAPS` columns of the constraint can be any type that can be compared for equality in a GiST index. By default, only range types are supported, but you can use other types by adding the btree_gist extension (which is the expected way to use this feature).

For the purpose of a unique constraint, null values are not considered equal, unless `NULLS NOT DISTINCT` is specified.

Each unique constraint should name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise, redundant unique constraints will be discarded.)

When establishing a unique constraint for a multi-level partition hierarchy, all the columns in the partition key of the target partitioned table, as well as those of all its descendant partitioned tables, must be included in the constraint definition.

Adding a unique constraint will automatically create a unique btree index on the column or group of columns used in the constraint. But if the constraint includes a `WITHOUT OVERLAPS` clause, it will use a GiST index. The created index has the same name as the unique constraint.

The optional `INCLUDE` clause adds to that index one or more columns that are simply "payload": uniqueness is not enforced on them, and the index cannot be searched on the basis of those columns. However they can be retrieved by an index-only scan. Note that although the constraint is not enforced on included columns, it still depends on them. Consequently, some operations on such columns (e.g., `DROP COLUMN`) can cause cascaded constraint and index deletion.

`PRIMARY KEY` (column constraint)
`PRIMARY KEY ( `*`column_name`*` [, ... ] [, `*`column_name`*` WITHOUT OVERLAPS ] )` [`INCLUDE ( `*`column_name`*` [, ...])`] (table constraint) #

The `PRIMARY KEY` constraint specifies that a column or columns of a table can contain only unique (non-duplicate), nonnull values. Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is different from the set of columns named by any unique constraint defined for the same table. (Otherwise, the unique constraint is redundant and will be discarded.)

`PRIMARY KEY` enforces the same data constraints as a combination of `UNIQUE` and `NOT NULL`. However, identifying a set of columns as the primary key also provides metadata about the design of the schema, since a primary key implies that other tables can rely on this set of columns as a unique identifier for rows.

When placed on a partitioned table, `PRIMARY KEY` constraints share the restrictions previously described for `UNIQUE` constraints.

Adding a `PRIMARY KEY` constraint will automatically create a unique btree index on the column or group of columns used in the constraint, or GiST if `WITHOUT OVERLAPS` was specified.

The optional `INCLUDE` clause adds to that index one or more columns that are simply "payload": uniqueness is not enforced on them, and the index cannot be searched on the basis of those columns. However they can be retrieved by an index-only scan. Note that although the constraint is not enforced on included columns, it still depends on them. Consequently, some operations on such columns (e.g., `DROP COLUMN`) can cause cascaded constraint and index deletion.

`EXCLUDE [ USING `*`index_method`*` ] ( `*`exclude_element`*` WITH `*`operator`*` [, ... ] )` *`index_parameters`*` [ WHERE ( `*`predicate`*` ) ]` #

The `EXCLUDE` clause defines an exclusion constraint, which guarantees that if any two rows are compared on the specified column(s) or expression(s) using the specified operator(s), not all of these comparisons will return TRUE. If all of the specified operators test for equality, this is equivalent to a `UNIQUE` constraint, although an ordinary unique constraint will be faster. However, exclusion constraints can specify constraints that are more general than simple equality. For example, you can specify a constraint that no two rows in the table contain overlapping circles (see Section 8.8) by using the && operator. The operator(s) are required to be commutative.

Exclusion constraints are implemented using an index that has the same name as the constraint, so each specified operator must be associated with an appropriate operator class (see Section 11.10) for the index access method *`index_method`*. Each *`exclude_element`* defines a column of the index, so it can optionally specify a collation, an operator class, operator class parameters, and/or ordering options; these are described fully under CREATE INDEX.

The access method must support `amgettuple` (see Chapter 63); at present this means GIN cannot be used. Although it's allowed, there is little point in using B-tree or hash indexes with an exclusion constraint, because this does nothing that an ordinary unique constraint doesn't do better. So in practice the access method will always be GiST or SP-GiST.

The *`predicate`* allows you to specify an exclusion constraint on a subset of the table; internally this creates a partial index. Note that parentheses are required around the predicate.

When establishing an exclusion constraint for a multi-level partition hierarchy, all the columns in the partition key of the target partitioned table, as well as those of all its descendant partitioned tables, must be included in the constraint definition. Additionally, those columns must be compared using the equality operator. These restrictions ensure that potentially-conflicting rows will exist in the same partition. The constraint may also refer to other columns which are not a part of any partition key, which can be compared using any appropriate operator.

`REFERENCES` *`reftable`* `[ (` *`refcolumn`* `) ] [ MATCH` *`matchtype`* `] [ ON DELETE` *`referential_action`* `] [ ON UPDATE` *`referential_action`* `]` (column constraint)
`FOREIGN KEY (` *`column_name`* `[, ... ] [, PERIOD` *`column_name`* `] ) REFERENCES` *`reftable`* `[ (` *`refcolumn`* `[, ... ] [, PERIOD` *`refcolumn`* `] ) ] [ MATCH` *`matchtype`* `] [ ON DELETE` *`referential_action`* `] [ ON UPDATE` *`referential_action`* `]` (table constraint) #

These clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If the *`refcolumn`* list is omitted, the primary key of the *`reftable`* is used. Otherwise, the *`refcolumn`* list must refer to the columns of a non-deferrable unique or primary key constraint or be the columns of a non-partial unique index.

If the last column is marked with `PERIOD`, it is treated in a special way. While the non-`PERIOD` columns are compared for equality (and there must be at least one of them), the `PERIOD` column is not. Instead, the constraint is considered satisfied if the referenced table has matching records (based on the non-`PERIOD` parts of the key) whose combined `PERIOD` values completely cover the referencing record's. In other words, the reference must have a referent for its entire duration. This column must be a range or multirange type. In addition, the referenced table must have a primary key or unique constraint declared with `WITHOUT OVERLAPS`. Finally, if the foreign key has a PERIOD *`column_name`* specification the corresponding *`refcolumn`*, if present, must also be marked `PERIOD`. If the *`refcolumn`* clause is omitted, and thus the reftable's primary key constraint chosen, the primary key must have its final column marked `WITHOUT OVERLAPS`.

For each pair of referencing and referenced column, if they are of a collatable data type, then the collations must either be both deterministic or else both the same. This ensures that both columns have a consistent notion of equality.

The user must have `REFERENCES` permission on the referenced table (either the whole table, or the specific referenced columns). The addition of a foreign key constraint requires a `SHARE ROW EXCLUSIVE` lock on the referenced table. Note that foreign key constraints cannot be defined between temporary tables and permanent tables.

A value inserted into the referencing column(s) is matched against the values of the referenced table and referenced columns using the given match type. There are three match types: `MATCH FULL`, `MATCH PARTIAL`, and `MATCH SIMPLE` (which is the default). `MATCH FULL` will not allow one column of a multicolumn foreign key to be null unless all foreign key columns are null; if they are all null, the row is not required to have a match in the referenced table. `MATCH SIMPLE` allows any of the foreign key columns to be null; if any of them are null, the row is not required to have a match in the referenced table. `MATCH PARTIAL` is not yet implemented. (Of course, `NOT NULL` constraints can be applied to the referencing column(s) to prevent these cases from arising.)

In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table's columns. The `ON DELETE` clause specifies the action to perform when a referenced row in the referenced table is being deleted. Likewise, the `ON UPDATE` clause specifies the action to perform when a referenced column in the referenced table is being updated to a new value. If the row is updated, but the referenced column is not actually changed, no action is done. Referential actions are executed as part of the data changing command, even if the constraint is deferred. There are the following possible actions for each clause:

`NO ACTION` #

> Produce an error if the deletion or update would create a foreign key constraint violation. If the constraint is deferred, this error will be produced at constraint check time if there still exist any referencing rows. This is the default action.

`RESTRICT` #

> Produce an error if a row to be deleted or updated matches a row in the referencing table. This prevents the action even if the state after the action would not violate the foreign key constraint. In particular, it prevents updates of referenced rows to values that are distinct but compare as equal. (But it does not prevent “no-op” updates that update a column to the same value.)
>
> In a temporal foreign key, this option is not supported.

`CASCADE` #

Delete any rows referencing the deleted row, or update the values of the referencing column(s) to the new values of the referenced columns, respectively.

In a temporal foreign key, this option is not supported.

`SET NULL [ ( column_name [, ... ] ) ]` #

Set all of the referencing columns, or a specified subset of the referencing columns, to null. A subset of columns can only be specified for `ON DELETE` actions.

In a temporal foreign key, this option is not supported.

`SET DEFAULT [ ( column_name [, ... ] ) ]` #

Set all of the referencing columns, or a specified subset of the referencing columns, to their default values. A subset of columns can only be specified for `ON DELETE` actions. (There must be a row in the referenced table matching the default values, if they are not null, or the operation will fail.)

In a temporal foreign key, this option is not supported.

If the referenced column(s) are changed frequently, it might be wise to add an index to the referencing column(s) so that referential actions associated with the foreign key constraint can be performed more efficiently.

`DEFERRABLE`
`NOT DEFERRABLE` #

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction (using the `SET CONSTRAINTS` command). `NOT DEFERRABLE` is the default. Currently, only `UNIQUE`, `PRIMARY KEY`, `EXCLUDE`, and `REFERENCES` (foreign key) constraints accept this clause. `NOT NULL` and `CHECK` constraints are not deferrable. Note that deferrable constraints cannot be used as conflict arbitrators in an `INSERT` statement that includes an `ON CONFLICT DO UPDATE` clause.

`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED` #

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is `INITIALLY IMMEDIATE`, it is checked after each statement. This is the default. If the constraint is `INITIALLY DEFERRED`, it is checked only at the end of the transaction. The constraint check time can be altered with the `SET CONSTRAINTS` command.

`ENFORCED`
`NOT ENFORCED` #

When the constraint is `ENFORCED`, then the database system will ensure that the constraint is satisfied, by checking the constraint at appropriate times (after each statement or at the end of the transaction, as appropriate). That is the default. If the constraint is `NOT ENFORCED`, the database system will not check the constraint. It is then up to the application code to ensure that the constraints are satisfied. The database system might still assume that the data actually satisfies the constraint for optimization decisions where this does not affect the correctness of the result.

`NOT ENFORCED` constraints can be useful as documentation if the actual checking of the constraint at run time is too expensive.

This is currently only supported for foreign key and `CHECK` constraints.

`USING` *`method`* #

This optional clause specifies the table access method to use to store the contents for the new table; the method needs be an access method of type `TABLE`. See Chapter 62 for more information. If this option is not specified, the default table access method is chosen for the new table. See default_table_access_method for more information.

When creating a partition, the table access method is the access method of its partitioned table, if set.

`WITH (` *`storage_parameter`* `[=` *`value`*`] [, ... ] )` #

This clause specifies optional storage parameters for a table or index; see Storage Parameters below for more information. For backward-compatibility the `WITH` clause for a table can also include `OIDS=FALSE` to specify that rows of the new table should not contain OIDs (object identifiers), `OIDS=TRUE` is not supported anymore.

`WITHOUT OIDS` #

This is backward-compatible syntax for declaring a table `WITHOUT OIDS`, creating a table `WITH OIDS` is not supported anymore.

`ON COMMIT` #

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

`PRESERVE ROWS` #

No special action is taken at the ends of transactions. This is the default behavior.

`DELETE ROWS` #

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic `TRUNCATE` is done at each commit. When used on a partitioned table, this is not cascaded to its partitions.

`DROP` #

The temporary table will be dropped at the end of the current transaction block. When used on a partitioned table, this action drops its partitions and when used on tables with inheritance children, it drops the dependent children.

`TABLESPACE` *`tablespace_name`* #

The *`tablespace_name`* is the name of the tablespace in which the new table is to be created. If not specified, default_tablespace is consulted, or temp_tablespaces if the table is temporary. For partitioned tables, since no storage is required for the table itself, the tablespace specified overrides `default_tablespace` as the default tablespace to use for any newly created partitions when no other tablespace is explicitly specified.

`USING INDEX TABLESPACE` *`tablespace_name`* #

This clause allows selection of the tablespace in which the index associated with a `UNIQUE`, `PRIMARY KEY`, or `EXCLUDE` constraint will be created. If not specified, default_tablespace is consulted, or temp_tablespaces if the table is temporary.

Storage Parameters

The `WITH` clause can specify *storage parameters* for tables, and for indexes associated with a `UNIQUE`, `PRIMARY KEY`, or `EXCLUDE` constraint. Storage parameters for indexes are documented in CREATE INDEX. The storage parameters currently available for tables are listed below. For many of these parameters, as shown, there is an additional parameter with the same name prefixed with `toast.`, which controls the behavior of the table's secondary TOAST table, if any (see Section 66.2 for more information about TOAST). If a table parameter value is set and the equivalent `toast.` parameter is not, the TOAST table will use the table's parameter value. Specifying these parameters for partitioned tables is not supported, but you may specify them for individual leaf partitions.

`fillfactor` (`integer`) #

The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page, and makes heap-only tuple updates more likely. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate. This parameter cannot be set for TOAST tables.

`toast_tuple_target` (`integer`) #

The toast_tuple_target specifies the minimum tuple length required before we try to compress and/or move long column values into TOAST tables, and is also the target length we try to reduce the length below once toasting begins. This affects columns marked as External (for move), Main (for compression), or Extended (for both) and applies only to new tuples. There is no effect on existing rows. By default this parameter is set to allow at least 4 tuples per block, which with the default block size will be 2040 bytes. Valid values are between 128 bytes and the (block size -

header), by default 8160 bytes. Changing this value may not be useful for very short or very long rows. Note that the default setting is often close to optimal, and it is possible that setting this parameter could have negative effects in some cases. This parameter cannot be set for TOAST tables.

`parallel_workers` (`integer`) #

This sets the number of workers that should be used to assist a parallel scan of this table. If not set, the system will determine a value based on the relation size. The actual number of workers chosen by the planner or by utility statements that use parallel scans may be less, for example due to the setting of max_worker_processes.

`autovacuum_enabled`, `toast.autovacuum_enabled` (`boolean`) #

Enables or disables the autovacuum daemon for a particular table. If true, the autovacuum daemon will perform automatic `VACUUM` and/or `ANALYZE` operations on this table following the rules discussed in Section 24.1.6. If false, this table will not be autovacuumed, except to prevent transaction ID wraparound. See Section 24.1.5 for more about wraparound prevention. Note that the autovacuum daemon does not run at all (except to prevent transaction ID wraparound) if the autovacuum parameter is false; setting individual tables' storage parameters does not override that. Therefore there is seldom much point in explicitly setting this storage parameter to `true`, only to `false`.

`vacuum_index_cleanup`, `toast.vacuum_index_cleanup` (`enum`) #

Forces or disables index cleanup when `VACUUM` is run on this table. The default value is `AUTO`. With `OFF`, index cleanup is disabled, with `ON` it is enabled, and with `AUTO` a decision is made dynamically, each time `VACUUM` runs. The dynamic behavior allows `VACUUM` to avoid needlessly scanning indexes to remove very few dead tuples. Forcibly disabling all index cleanup can speed up `VACUUM` very significantly, but may also lead to severely bloated indexes if table modifications are frequent. The `INDEX_CLEANUP` parameter of `VACUUM`, if specified, overrides the value of this option.

`vacuum_truncate`, `toast.vacuum_truncate` (`boolean`) #

Per-table value for vacuum_truncate parameter. The `TRUNCATE` parameter of `VACUUM`, if specified, overrides the value of this option.

`autovacuum_vacuum_threshold`, `toast.autovacuum_vacuum_threshold` (`integer`) #

Per-table value for autovacuum_vacuum_threshold parameter.

`autovacuum_vacuum_max_threshold`, `toast.autovacuum_vacuum_max_threshold` (`integer`) #

Per-table value for autovacuum_vacuum_max_threshold parameter.

`autovacuum_vacuum_scale_factor`, `toast.autovacuum_vacuum_scale_factor` (`floating point`) #

Per-table value for autovacuum_vacuum_scale_factor parameter.

`autovacuum_vacuum_insert_threshold`, `toast.autovacuum_vacuum_insert_threshold` (`integer`) #

Per-table value for autovacuum_vacuum_insert_threshold parameter. The special value of -1 may be used to disable insert vacuums on the table.

`autovacuum_vacuum_insert_scale_factor`, `toast.autovacuum_vacuum_insert_scale_factor` (`floating point`) #

Per-table value for autovacuum_vacuum_insert_scale_factor parameter.

`autovacuum_analyze_threshold` (`integer`) #

Per-table value for autovacuum_analyze_threshold parameter.

`autovacuum_analyze_scale_factor` (`floating point`) #

Per-table value for autovacuum_analyze_scale_factor parameter.

`autovacuum_vacuum_cost_delay`, `toast.autovacuum_vacuum_cost_delay` (`floating point`) #

Per-table value for autovacuum_vacuum_cost_delay parameter.

`autovacuum_vacuum_cost_limit`, `toast.autovacuum_vacuum_cost_limit` (`integer`) #

Per-table value for autovacuum_vacuum_cost_limit parameter.

`autovacuum_freeze_min_age`, `toast.autovacuum_freeze_min_age` (`integer`) #

Per-table value for vacuum_freeze_min_age parameter. Note that autovacuum will ignore per-table `autovacuum_freeze_min_age` parameters that are larger than half the system-wide autovacuum_freeze_max_age setting.

`autovacuum_freeze_max_age`, `toast.autovacuum_freeze_max_age` (`integer`) #

Per-table value for autovacuum_freeze_max_age parameter. Note that autovacuum will ignore per-table `autovacuum_freeze_max_age` parameters that are larger than the system-wide setting (it can only be set smaller).

`autovacuum_freeze_table_age`, `toast.autovacuum_freeze_table_age` (`integer`) #

Per-table value for vacuum_freeze_table_age parameter.

`autovacuum_multixact_freeze_min_age`, `toast.autovacuum_multixact_freeze_min_age` (`integer`) #

Per-table value for vacuum_multixact_freeze_min_age parameter. Note that autovacuum will ignore per-table `autovacuum_multixact_freeze_min_age` parameters that are larger than half the system-wide autovacuum_multixact_freeze_max_age setting.

`autovacuum_multixact_freeze_max_age`, `toast.autovacuum_multixact_freeze_max_age` (`integer`) #

Per-table value for autovacuum_multixact_freeze_max_age parameter. Note that autovacuum will ignore per-table `autovacuum_multixact_freeze_max_age` parameters that are larger than the system-wide setting (it can only be set smaller).

`autovacuum_multixact_freeze_table_age`, `toast.autovacuum_multixact_freeze_table_age` (`integer`) #

Per-table value for vacuum_multixact_freeze_table_age parameter.

`log_autovacuum_min_duration`, `toast.log_autovacuum_min_duration` (`integer`) #

Per-table value for log_autovacuum_min_duration parameter.

`vacuum_max_eager_freeze_failure_rate`, `toast.vacuum_max_eager_freeze_failure_rate` (`floating point`) #

Per-table value for vacuum_max_eager_freeze_failure_rate parameter.

`user_catalog_table` (`boolean`) #

Declare the table as an additional catalog table for purposes of logical replication. See Section 47.6.2 for details. This parameter cannot be set for TOAST tables.

Notes

PostgreSQL automatically creates an index for each unique constraint and primary key constraint to enforce uniqueness. Thus, it is not necessary to create an index explicitly for primary key columns. (See CREATE INDEX for more information.)

Unique constraints and primary keys are not inherited in the current implementation. This makes the combination of inheritance and unique constraints rather dysfunctional.

A table cannot have more than 1600 columns. (In practice, the effective limit is usually lower because of tuple-length constraints.)

Examples

Create table `films` and table `distributors`:

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

Create a table with a 2-dimensional array:

```
CREATE TABLE array_int (
    vector  int[][]
);
```

Define a unique table constraint for the table `films`. Unique table constraints can be defined on one or more columns of the table:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Define a check column constraint:

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

Define a check table constraint:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

Define a primary key table constraint for the table `films`:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Define a primary key constraint for table `distributors`. The following two examples are equivalent, the first using the table constraint syntax, the second the column constraint syntax:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

Assign a literal constant default value for the column `name`, arrange for the default value of column `did` to be generated by selecting the next value of a sequence object, and make the default value of `modtime` be the time at which the row is inserted:

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

Define two `NOT NULL` column constraints on the table `distributors`, one of which is explicitly given a name:

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

Define a unique constraint for the `name` column:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

The same, specified as a table constraint:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

Create the same table, specifying 70% fill factor for both the table and its unique index:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

Create table `circles` with an exclusion constraint that prevents any two circles from overlapping:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Create table `cinemas` in tablespace `diskvol1`:

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

Create a composite type and a typed table:

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

Create a range partitioned table:

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

Create a range partitioned table with multiple columns in the partition key:

```
CREATE TABLE measurement_year_month (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH FROM
logdate));
```

Create a list partitioned table:

```
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));
```

Create a hash partitioned table:

```
CREATE TABLE orders (
    order_id     bigint not null,
    cust_id      bigint not null,
    status       text
) PARTITION BY HASH (order_id);
```

Create partition of a range partitioned table:

```
CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
    unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

Create a few partitions of a range partitioned table with multiple columns in the partition key:

```
CREATE TABLE measurement_ym_older
    PARTITION OF measurement_year_month
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

CREATE TABLE measurement_ym_y2016m11
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 11) TO (2016, 12);

CREATE TABLE measurement_ym_y2016m12
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 12) TO (2017, 01);

CREATE TABLE measurement_ym_y2017m01
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2017, 01) TO (2017, 02);
```

Create partition of a list partitioned table:

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

Create partition of a list partitioned table that is itself further partitioned and then add a partition to it:

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);

CREATE TABLE cities_ab_10000_to_100000
    PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

Create partitions of a hash partitioned table:

```
CREATE TABLE orders_p1 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Create a default partition:

```
CREATE TABLE cities_partdef
    PARTITION OF cities DEFAULT;
```

Compatibility

The `CREATE TABLE` command conforms to the SQL standard, with exceptions listed below.

Temporary Tables

Although the syntax of `CREATE TEMPORARY TABLE` resembles that of the SQL standard, the effect is not the same. In the standard, temporary tables are defined just once and automatically exist (starting with empty contents) in every session that needs them. PostgreSQL instead requires each session to issue its own `CREATE TEMPORARY TABLE` command for each temporary table to be used. This allows different sessions to use the same temporary table name for different purposes, whereas the standard's approach constrains all instances of a given temporary table name to have the same table structure.

The standard's definition of the behavior of temporary tables is widely ignored. PostgreSQL's behavior on this point is similar to that of several other SQL databases.

The SQL standard also distinguishes between global and local temporary tables, where a local temporary table has a separate set of contents for each SQL module within each session, though its definition is still shared across sessions. Since PostgreSQL does not support SQL modules, this distinction is not relevant in PostgreSQL.

For compatibility's sake, PostgreSQL will accept the `GLOBAL` and `LOCAL` keywords in a temporary table declaration, but they currently have no effect. Use of these keywords is discouraged, since future versions of PostgreSQL might adopt a more standard-compliant interpretation of their meaning.

The `ON COMMIT` clause for temporary tables also resembles the SQL standard, but has some differences. If the `ON COMMIT` clause is omitted, SQL specifies that the default behavior is `ON COMMIT DELETE ROWS`. However, the default behavior in PostgreSQL is `ON COMMIT PRESERVE ROWS`. The `ON COMMIT DROP` option does not exist in SQL.

Non-Deferred Uniqueness Constraints

When a `UNIQUE` or `PRIMARY KEY` constraint is not deferrable, PostgreSQL checks for uniqueness immediately whenever a row is inserted or modified. The SQL standard says that uniqueness should be enforced only at the end of the statement; this makes a difference when, for example, a single command updates multiple key values. To obtain standard-compliant behavior, declare the constraint as `DEFERRABLE` but not deferred (i.e., `INITIALLY IMMEDIATE`). Be aware that this can be significantly slower than immediate uniqueness checking.

Column Check Constraints

The SQL standard says that `CHECK` column constraints can only refer to the column they apply to; only `CHECK` table constraints can refer to multiple columns. PostgreSQL does not enforce this restriction; it treats column and table check constraints alike.

`EXCLUDE` Constraint

The `EXCLUDE` constraint type is a PostgreSQL extension.

Foreign Key Constraints

The ability to specify column lists in the foreign key actions `SET DEFAULT` and `SET NULL` is a PostgreSQL extension.

It is a PostgreSQL extension that a foreign key constraint may reference columns of a unique index instead of columns of a primary key or unique constraint.

`NULL` “Constraint”

The `NULL` “constraint” (actually a non-constraint) is a PostgreSQL extension to the SQL standard that is included for compatibility with some other database systems (and for symmetry with the `NOT NULL` constraint). Since it is the default for any column, its presence is simply noise.

Constraint Naming

The SQL standard says that table and domain constraints must have names that are unique across the schema containing the table or domain. PostgreSQL is laxer: it only requires constraint names to be unique across the constraints attached to a particular table or domain. However, this extra freedom does not exist for index-based constraints (`UNIQUE`, `PRIMARY KEY`, and `EXCLUDE` constraints), because the associated index is named the same as the constraint, and index names must be unique across all relations within the same schema.

Inheritance

Multiple inheritance via the `INHERITS` clause is a PostgreSQL language extension. SQL:1999 and later define single inheritance using a different syntax and different semantics. SQL:1999-style inheritance is not yet supported by PostgreSQL.

Zero-Column Tables

PostgreSQL allows a table of no columns to be created (for example, `CREATE TABLE foo();`). This is an extension from the SQL standard, which does not allow zero-column tables. Zero-column tables are not in themselves very useful, but disallowing them creates odd special cases for `ALTER TABLE DROP COLUMN`, so it seems cleaner to ignore this spec restriction.

Multiple Identity Columns

PostgreSQL allows a table to have more than one identity column. The standard specifies that a table can have at most one identity column. This is relaxed mainly to give more flexibility for doing schema changes or migrations. Note that the `INSERT` command supports only one override clause that applies to the entire statement, so having multiple identity columns with different behaviors is not well supported.

Generated Columns

The options `STORED` and `VIRTUAL` are not standard but are also used by other SQL implementations. The SQL standard does not specify the storage of generated columns.

`LIKE` Clause

While a `LIKE` clause exists in the SQL standard, many of the options that PostgreSQL accepts for it are not in the standard, and some of the standard's options are not implemented by PostgreSQL.

`WITH` Clause

The `WITH` clause is a PostgreSQL extension; storage parameters are not in the standard.

Tablespaces

The PostgreSQL concept of tablespaces is not part of the standard. Hence, the clauses `TABLESPACE` and `USING INDEX TABLESPACE` are extensions.

Typed Tables

Typed tables implement a subset of the SQL standard. According to the standard, a typed table has columns corresponding to the underlying composite type as well as one other column that is the “self-referencing column”. PostgreSQL does not support self-referencing columns explicitly.

`PARTITION BY` Clause

The `PARTITION BY` clause is a PostgreSQL extension.

`PARTITION OF` Clause

The `PARTITION OF` clause is a PostgreSQL extension.

Prev Up Next
CREATE SUBSCRIPTION Home CREATE TABLE AS

Submit correction

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use this form to report a documentation issue.