NAME

ALTER_TABLE − change the definition of a table

SYNOPSIS

ALTER TABLE [IF EXISTS] [ONLY] *name* [*]
action [, ...]
ALTER TABLE [IF EXISTS] [ONLY] *name* [*]
RENAME [COLUMN] *column_name* TO *new_column_name*
ALTER TABLE [IF EXISTS] [ONLY] *name* [*]
RENAME CONSTRAINT *constraint_name* TO *new_constraint_name*
ALTER TABLE [IF EXISTS] *name*
RENAME TO *new_name*
ALTER TABLE [IF EXISTS] *name*
SET SCHEMA *new_schema*
ALTER TABLE ALL IN TABLESPACE *name* [OWNED BY *role_name* [, ...]]
SET TABLESPACE *new_tablespace* [NOWAIT]
ALTER TABLE [IF EXISTS] *name*
ATTACH PARTITION *partition_name* { FOR VALUES *partition_bound_spec* | DEFAULT }
ALTER TABLE [IF EXISTS] *name*
DETACH PARTITION *partition_name* [CONCURRENTLY | FINALIZE]

where *action* is one of:

ADD [COLUMN] [IF NOT EXISTS] *column_name data_type* [COLLATE *collation*] [*column_constraint* [...]]
DROP [COLUMN] [IF EXISTS] *column_name* [RESTRICT | CASCADE]
ALTER [COLUMN] *column_name* [SET DATA] TYPE *data_type* [COLLATE *collation*] [USING *expression*]
ALTER [COLUMN] *column_name* SET DEFAULT *expression*
ALTER [COLUMN] *column_name* DROP DEFAULT
ALTER [COLUMN] *column_name* { SET | DROP } NOT NULL
ALTER [COLUMN] *column_name* SET EXPRESSION AS (*expression*)
ALTER [COLUMN] *column_name* DROP EXPRESSION [IF EXISTS]
ALTER [COLUMN] *column_name* ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(*sequence_optio*
ALTER [COLUMN] *column_name* { SET GENERATED { ALWAYS | BY DEFAULT } | SET *sequence_option* | RESTA
ALTER [COLUMN] *column_name* DROP IDENTITY [IF EXISTS]
ALTER [COLUMN] *column_name* SET STATISTICS { *integer* | DEFAULT }
ALTER [COLUMN] *column_name* SET (*attribute_option* = *value* [, ...])
ALTER [COLUMN] *column_name* RESET (*attribute_option* [, ...])
ALTER [COLUMN] *column_name* SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }
ALTER [COLUMN] *column_name* SET COMPRESSION *compression_method*
ADD *table_constraint* [NOT VALID]
ADD *table_constraint_using_index*
ALTER CONSTRAINT *constraint_name* [DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIAL
ALTER CONSTRAINT *constraint_name* [INHERIT | NO INHERIT]
VALIDATE CONSTRAINT *constraint_name*
DROP CONSTRAINT [IF EXISTS] *constraint_name* [RESTRICT | CASCADE]
DISABLE TRIGGER [*trigger_name* | ALL | USER]
ENABLE TRIGGER [*trigger_name* | ALL | USER]
ENABLE REPLICA TRIGGER *trigger_name*
ENABLE ALWAYS TRIGGER *trigger_name*
DISABLE RULE *rewrite_rule_name*
ENABLE RULE *rewrite_rule_name*
ENABLE REPLICA RULE *rewrite_rule_name*
ENABLE ALWAYS RULE *rewrite_rule_name*
DISABLE ROW LEVEL SECURITY
ENABLE ROW LEVEL SECURITY

```
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    SET ACCESS METHOD { new_access_method | DEFAULT }
    SET TABLESPACE new_tablespace
    SET { LOGGED | UNLOGGED }
    SET ( storage_parameter [= value] [, ... ] )
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OF type_name
    NOT OF
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }
```

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 )
```

and *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 EN
```

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_paramet
  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
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE referential_action ] [ ON UPDATE referential_
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ] [ ENFORCED | NOT EN
```

and *table_constraint_using_index* is:

```
    [ CONSTRAINT constraint_name ]
```

{ UNIQUE | PRIMARY KEY } USING INDEX *index_name*
[DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]

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] [

referential_action in a FOREIGN KEY/REFERENCES constraint is:

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

DESCRIPTION

ALTER TABLE changes the definition of an existing table. There are several subforms described below. Note that the lock level required may differ for each subform. An ACCESS EXCLUSIVE lock is acquired unless explicitly noted. When multiple subcommands are given, the lock acquired will be the strictest one required by any subcommand.

ADD [COLUMN] [IF NOT EXISTS]
: This form adds a new column to the table, using the same syntax as **CREATE TABLE**. If IF NOT EXISTS is specified and a column already exists with this name, no error is thrown.

DROP [COLUMN] [IF EXISTS]
: This form drops a column from a table. Indexes and table constraints involving the column will be automatically dropped as well. Multivariate statistics referencing the dropped column will also be removed if the removal of the column would cause the statistics to contain data for only a single column. You will need to say CASCADE if anything outside the table depends on the column, for example, foreign key references or views. If IF EXISTS is specified and the column does not exist, no error is thrown. In this case a notice is issued instead.

SET DATA TYPE
: This form changes the type of a column of a table. Indexes and simple table constraints involving the column will be automatically converted to use the new column type by reparsing the originally supplied expression. The optional COLLATE clause specifies a collation for the new column; if omitted, the collation is the default for the new column type. The optional USING clause specifies how to compute the new column value from the old; if omitted, the default conversion is the same as an assignment cast from old data type to new. A USING clause must be provided if there is no implicit or assignment cast from old to new type.

  When this form is used, the column's statistics are removed, so running **ANALYZE** on the table afterwards is recommended. For a virtual generated column, **ANALYZE** is not necessary because such columns never have statistics.

SET/DROP DEFAULT
: These forms set or remove the default value for a column (where removal is equivalent to setting the default value to NULL). The new default value will only apply in subsequent **INSERT** or **UPDATE** commands; it does not cause rows already in the table to change.

SET/DROP NOT NULL
: These forms change whether a column is marked to allow null values or to reject null values.

  SET NOT NULL may only be applied to a column provided none of the records in the table contain a NULL value for the column. Ordinarily this is checked during the ALTER TABLE by scanning the

entire table, unless NOT VALID is specified; however, if a valid CHECK constraint exists (and is not dropped in the same command) which proves no NULL can exist, then the table scan is skipped. If a column has an invalid not−null constraint, SET NOT NULL validates it.

If this table is a partition, one cannot perform DROP NOT NULL on a column if it is marked NOT NULL in the parent table. To drop the NOT NULL constraint from all the partitions, perform DROP NOT NULL on the parent table. Even if there is no NOT NULL constraint on the parent, such a constraint can still be added to individual partitions, if desired; that is, the children can disallow nulls even if the parent allows them, but not the other way around. It is also possible to drop the NOT NULL constraint from ONLY the parent table, which does not remove it from the children.

SET EXPRESSION AS
: This form replaces the expression of a generated column. Existing data in a stored generated column is rewritten and all the future changes will apply the new generation expression.

  When this form is used on a stored generated column, its statistics are removed, so running **ANALYZE** on the table afterwards is recommended. For a virtual generated column, **ANALYZE** is not necessary because such columns never have statistics.

DROP EXPRESSION [IF EXISTS]
: This form turns a stored generated column into a normal base column. Existing data in the columns is retained, but future changes will no longer apply the generation expression.

  This form is currently only supported for stored generated columns (not virtual ones).

  If DROP EXPRESSION IF EXISTS is specified and the column is not a generated column, no error is thrown. In this case a notice is issued instead.

ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
SET GENERATED { ALWAYS | BY DEFAULT }
DROP IDENTITY [IF EXISTS]
: These forms change whether a column is an identity column or change the generation attribute of an existing identity column. See **CREATE TABLE** for details. Like SET DEFAULT, these forms only affect the behavior of subsequent **INSERT** and **UPDATE** commands; they do not cause rows already in the table to change.

  If DROP IDENTITY IF EXISTS is specified and the column is not an identity column, no error is thrown. In this case a notice is issued instead.

SET *sequence_option*
RESTART
: These forms alter the sequence that underlies an existing identity column. *sequence_option* is an option supported by **ALTER SEQUENCE** such as INCREMENT BY.

SET STATISTICS
: This form sets the per−column statistics−gathering target for subsequent **ANALYZE** operations. The target can be set in the range 0 to 10000. Set it to DEFAULT to revert to using the system default statistics target (default_statistics_target). (Setting to a value of −1 is an obsolete way spelling to get the same outcome.) For more information on the use of statistics by the PostgreSQL query planner, refer to Section 14.2.

  SET STATISTICS acquires a SHARE UPDATE EXCLUSIVE lock.

SET (*attribute_option* = *value* [, ...])
RESET (*attribute_option* [, ...])
: This form sets or resets per−attribute options. Currently, the only defined per−attribute options are n_distinct and n_distinct_inherited, which override the number−of−distinct−values estimates made by subsequent **ANALYZE** operations. n_distinct affects the statistics for the table itself, while

n_distinct_inherited affects the statistics gathered for the table plus its inheritance children, and for the statistics gathered for partitioned tables. When the value specified is a positive value, the query planner will assume that the column contains exactly the specified number of distinct nonnull values. Fractional values may also be specified by using values below 0 and above or equal to −1. This instructs the query planner to estimate the number of distinct values by multiplying the absolute value of the specified number by the estimated number of rows in the table. For example, a value of −1 implies that all values in the column are distinct, while a value of −0.5 implies that each value appears twice on average. This can be useful when the size of the table changes over time. For more information on the use of statistics by the PostgreSQL query planner, refer to Section 14.2.

Changing per−attribute options acquires a SHARE UPDATE EXCLUSIVE lock.

SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }
This form sets the storage mode for a 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. Note that ALTER TABLE ... SET STORAGE doesn't itself change anything in the table; it just sets the strategy to be pursued during future table updates. See Section 66.2 for more information.

SET COMPRESSION *compression_method*
This form sets the compression method for a column, determining how values inserted in future will be compressed (if the storage mode permits compression at all). This does not cause the table to be rewritten, so existing data may still be compressed with other compression methods. If the table is restored with pg_restore, then all values are rewritten with the configured compression method. However, when data is inserted from another relation (for example, by **INSERT ... SELECT**), values from the source table are not necessarily detoasted, so any previously compressed data may retain its existing compression method, rather than being recompressed with the compression method of the target column. 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, which selects the default behavior of consulting the default_toast_compression setting at the time of data insertion to determine the method to use.

ADD *table_constraint* [NOT VALID]
This form adds a new constraint to a table using the same constraint syntax as **CREATE TABLE**, plus the option NOT VALID, which is currently only allowed for foreign−key, CHECK, and not−null constraints.

Normally, this form will cause a scan of the table to verify that all existing rows in the table satisfy the new constraint. But if the NOT VALID option is used, this potentially−lengthy scan is skipped. The constraint will still be applied against subsequent inserts or updates (that is, they'll fail unless there is a matching row in the referenced table, in the case of foreign keys, or they'll fail unless the new row matches the specified check condition). But the database will not assume that the constraint holds for all rows in the table, until it is validated by using the VALIDATE CONSTRAINT option. See Notes below for more information about using the NOT VALID option.

Although most forms of ADD *table_constraint* require an ACCESS EXCLUSIVE lock, ADD FOREIGN KEY requires only a SHARE ROW EXCLUSIVE lock. Note that ADD FOREIGN KEY also acquires a SHARE ROW EXCLUSIVE lock on the referenced table, in addition to the lock on the table on which the constraint is declared.

Additional restrictions apply when unique or primary key constraints are added to partitioned tables;

see **CREATE TABLE**.

ADD *table_constraint_using_index*

This form adds a new PRIMARY KEY or UNIQUE constraint to a table based on an existing unique index. All the columns of the index will be included in the constraint.

The index cannot have expression columns nor be a partial index. Also, it must be a b−tree index with default sort ordering. These restrictions ensure that the index is equivalent to one that would be built by a regular ADD PRIMARY KEY or ADD UNIQUE command.

If PRIMARY KEY is specified, and the index's columns are not already marked NOT NULL, then this command will attempt to do ALTER COLUMN SET NOT NULL against each such column. That requires a full table scan to verify the column(s) contain no nulls. In all other cases, this is a fast operation.

If a constraint name is provided then the index will be renamed to match the constraint name. Otherwise the constraint will be named the same as the index.

After this command is executed, the index is “owned” by the constraint, in the same way as if the index had been built by a regular ADD PRIMARY KEY or ADD UNIQUE command. In particular, dropping the constraint will make the index disappear too.

This form is not currently supported on partitioned tables.

Note

Adding a constraint using an existing index can be helpful in situations where a new constraint needs to be added without blocking table updates for a long time. To do that, create the index using **CREATE UNIQUE INDEX CONCURRENTLY**, and then convert it to a constraint using this syntax. See the example below.

ALTER CONSTRAINT

This form alters the attributes of a constraint that was previously created. Currently only foreign key constraints may be altered in this fashion, but see below.

ALTER CONSTRAINT ... INHERIT

ALTER CONSTRAINT ... NO INHERIT

These forms modify a inheritable constraint so that it becomes not inheritable, or vice−versa. Only not−null constraints may be altered in this fashion at present. In addition to changing the inheritability status of the constraint, in the case where a non−inheritable constraint is being marked inheritable, if the table has children, an equivalent constraint will be added to them. If marking an inheritable constraint as non−inheritable on a table with children, then the corresponding constraint on children will be marked as no longer inherited, but not removed.

VALIDATE CONSTRAINT

This form validates a foreign key, check, or not−null constraint that was previously created as NOT VALID, by scanning the table to ensure there are no rows for which the constraint is not satisfied. If the constraint was set to NOT ENFORCED, an error is thrown. Nothing happens if the constraint is already marked valid. (See Notes below for an explanation of the usefulness of this command.)

This command acquires a SHARE UPDATE EXCLUSIVE lock.

DROP CONSTRAINT [IF EXISTS]

This form drops the specified constraint on a table, along with any index underlying the constraint. If IF EXISTS is specified and the constraint does not exist, no error is thrown. In this case a notice is issued instead.

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

These forms configure the firing of trigger(s) belonging to the table. A disabled trigger is still known to the system, but is not executed when its triggering event occurs. (For a deferred trigger, the enable

status is checked when the event occurs, not when the trigger function is actually executed.) One can disable or enable a single trigger specified by name, or all triggers on the table, or only user triggers (this option excludes internally generated constraint triggers, such as those that are used to implement foreign key constraints or deferrable uniqueness and exclusion constraints). Disabling or enabling internally generated constraint triggers requires superuser privileges; it should be done with caution since of course the integrity of the constraint cannot be guaranteed if the triggers are not executed.

The trigger firing mechanism is also affected by the configuration variable session_replication_role. Simply enabled triggers (the default) will fire when the replication role is “origin” (the default) or “local”. Triggers configured as ENABLE REPLICA will only fire if the session is in “replica” mode, and triggers configured as ENABLE ALWAYS will fire regardless of the current replication role.

The effect of this mechanism is that in the default configuration, triggers do not fire on replicas. This is useful because if a trigger is used on the origin to propagate data between tables, then the replication system will also replicate the propagated data; so the trigger should not fire a second time on the replica, because that would lead to duplication. However, if a trigger is used for another purpose such as creating external alerts, then it might be appropriate to set it to ENABLE ALWAYS so that it is also fired on replicas.

When this command is applied to a partitioned table, the states of corresponding clone triggers in the partitions are updated too, unless ONLY is specified.

This command acquires a SHARE ROW EXCLUSIVE lock.

DISABLE/ENABLE [REPLICA | ALWAYS] RULE

These forms configure the firing of rewrite rules belonging to the table. A disabled rule is still known to the system, but is not applied during query rewriting. The semantics are as for disabled/enabled triggers. This configuration is ignored for ON SELECT rules, which are always applied in order to keep views working even if the current session is in a non−default replication role.

The rule firing mechanism is also affected by the configuration variable session_replication_role, analogous to triggers as described above.

DISABLE/ENABLE ROW LEVEL SECURITY

These forms control the application of row security policies belonging to the table. If enabled and no policies exist for the table, then a default−deny policy is applied. Note that policies can exist for a table even if row−level security is disabled. In this case, the policies will *not* be applied and the policies will be ignored. See also **CREATE POLICY**.

NO FORCE/FORCE ROW LEVEL SECURITY

These forms control the application of row security policies belonging to the table when the user is the table owner. If enabled, row−level security policies will be applied when the user is the table owner. If disabled (the default) then row−level security will not be applied when the user is the table owner. See also **CREATE POLICY**.

CLUSTER ON

This form selects the default index for future **CLUSTER** operations. It does not actually re−cluster the table.

Changing cluster options acquires a SHARE UPDATE EXCLUSIVE lock.

SET WITHOUT CLUSTER

This form removes the most recently used **CLUSTER** index specification from the table. This affects future cluster operations that don't specify an index.

Changing cluster options acquires a SHARE UPDATE EXCLUSIVE lock.

SET WITHOUT OIDS

Backward−compatible syntax for removing the oid system column. As oid system columns cannot be added anymore, this never has an effect.

SET ACCESS METHOD
This form changes the access method of the table by rewriting it using the indicated access method; specifying DEFAULT selects the access method set as the default_table_access_method configuration parameter. See Chapter 62 for more information.

When applied to a partitioned table, there is no data to rewrite, but partitions created afterwards will default to the given access method unless overridden by a USING clause. Specifying *DEFAULT* removes a previous value, causing future partitions to default to *default_table_access_method*.

SET TABLESPACE
This form changes the table's tablespace to the specified tablespace and moves the data file(s) associated with the table to the new tablespace. Indexes on the table, if any, are not moved; but they can be moved separately with additional SET TABLESPACE commands. When applied to a partitioned table, nothing is moved, but any partitions created afterwards with **CREATE TABLE PARTITION OF** will use that tablespace, unless overridden by a TABLESPACE clause.

All tables in the current database in a tablespace can be moved by using the ALL IN TABLESPACE form, which will lock all tables to be moved first and then move each one. This form also supports OWNED BY, which will only move tables owned by the roles specified. If the NOWAIT option is specified then the command will fail if it is unable to acquire all of the locks required immediately. Note that system catalogs are not moved by this command; use **ALTER DATABASE** or explicit **ALTER TABLE** invocations instead if desired. The information_schema relations are not considered part of the system catalogs and will be moved. See also **CREATE TABLESPACE**.

SET { LOGGED | UNLOGGED }
This form changes the table from unlogged to logged or vice−versa (see UNLOGGED). It cannot be applied to a temporary table.

This also changes the persistence of any sequences linked to the table (for identity or serial columns). However, it is also possible to change the persistence of such sequences separately.

This form is not supported for partitioned tables.

SET (*storage_parameter* [= *value*] [, ...])
This form changes one or more storage parameters for the table. See Storage Parameters in the **CREATE TABLE** documentation for details on the available parameters. Note that the table contents will not be modified immediately by this command; depending on the parameter you might need to rewrite the table to get the desired effects. That can be done with **VACUUM FULL**, **CLUSTER** or one of the forms of **ALTER TABLE** that forces a table rewrite. For planner related parameters, changes will take effect from the next time the table is locked so currently executing queries will not be affected.

SHARE UPDATE EXCLUSIVE lock will be taken for fillfactor, toast and autovacuum storage parameters, as well as the planner parameter *parallel_workers*.

RESET (*storage_parameter* [, ...])
This form resets one or more storage parameters to their defaults. As with SET, a table rewrite might be needed to update the table entirely.

INHERIT *parent_table*
This form adds the target table as a new child of the specified parent table. Subsequently, queries against the parent will include records of the target table. To be added as a child, the target table must already contain all the same columns as the parent (it could have additional columns, too). The columns must have matching data types.

In addition, all CHECK and NOT NULL constraints on the parent must also exist on the child, except those marked non−inheritable (that is, created with ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT), which are ignored. All child−table constraints matched must not be marked non−inheritable. Currently UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints are not considered, but this might change in the future.

NO INHERIT *parent_table*
This form removes the target table from the list of children of the specified parent table. Queries against the parent table will no longer include records drawn from the target table.

OF *type_name*
This form links the table to a composite type as though **CREATE TABLE OF** had formed it. The table's list of column names and types must precisely match that of the composite type. The table must not inherit from any other table. These restrictions ensure that **CREATE TABLE OF** would permit an equivalent table definition.

NOT OF
This form dissociates a typed table from its type.

OWNER TO
This form changes the owner of the table, sequence, view, materialized view, or foreign table to the specified user.

REPLICA IDENTITY
This form changes the information which is written to the write−ahead log to identify rows which are updated or deleted. In most cases, the old value of each column is only logged if it differs from the new value; however, if the old value is stored externally, it is always logged regardless of whether it changed. This option has no effect except when logical replication is in use.

DEFAULT
Records the old values of the columns of the primary key. This is the default for non−system tables. When there is no primary key, the behavior is the same as NOTHING.

USING INDEX *index_name*
Records the old values of the columns covered by the named index, that must be unique, not partial, not deferrable, and include only columns marked NOT NULL. If this index is dropped, the behavior is the same as NOTHING.

FULL
Records the old values of all columns in the row.

NOTHING
Records no information about the old row. This is the default for system tables.

RENAME
The RENAME forms change the name of a table (or an index, sequence, view, materialized view, or foreign table), the name of an individual column in a table, or the name of a constraint of the table. When renaming a constraint that has an underlying index, the index is renamed as well. There is no effect on the stored data.

SET SCHEMA
This form moves the table into another schema. Associated indexes, constraints, and sequences owned by table columns are moved as well.

ATTACH PARTITION *partition_name* { FOR VALUES *partition_bound_spec* | DEFAULT }
This form attaches an existing table (which might itself be partitioned) as a partition of the target table. The table can be attached as a partition for specific values using FOR VALUES or as a default partition by using DEFAULT. For each index in the target table, a corresponding one will be created in the attached table; or, if an equivalent index already exists, it will be attached to the target table's index, as if **ALTER INDEX ATTACH PARTITION** had been executed. Note that if the existing table is a foreign table, it is currently not allowed to attach the table as a partition of the target table if there are UNIQUE indexes on the target table. (See also CREATE FOREIGN TABLE

(**CREATE_FOREIGN_TABLE**(7)).) For each user−defined row−level trigger that exists in the target table, a corresponding one is created in the attached table.

A partition using FOR VALUES uses same syntax for *partition_bound_spec* as **CREATE TABLE**. The partition bound specification must correspond to the partitioning strategy and partition key of the target table. The table to be attached must have all the same columns as the target table and no more; moreover, the column types must also match. Also, it must have all the NOT NULL and CHECK constraints of the target table, not marked NO INHERIT. Currently FOREIGN KEY constraints are not considered. UNIQUE and PRIMARY KEY constraints from the parent table will be created in the partition, if they don't already exist.

If the new partition is a regular table, a full table scan is performed to check that existing rows in the table do not violate the partition constraint. It is possible to avoid this scan by adding a valid CHECK constraint to the table that allows only rows satisfying the desired partition constraint before running this command. The CHECK constraint will be used to determine that the table need not be scanned to validate the partition constraint. This does not work, however, if any of the partition keys is an expression and the partition does not accept NULL values. If attaching a list partition that will not accept NULL values, also add a NOT NULL constraint to the partition key column, unless it's an expression.

If the new partition is a foreign table, nothing is done to verify that all the rows in the foreign table obey the partition constraint. (See the discussion in CREATE FOREIGN TABLE (**CREATE_FOREIGN_TABLE**(7)) about constraints on the foreign table.)

When a table has a default partition, defining a new partition changes the partition constraint for the default partition. The default partition can't contain any rows that would need to be moved to the new partition, and will be scanned to verify that none are present. This scan, like the scan of the new partition, can be avoided if an appropriate CHECK constraint is present. Also like the scan of the new partition, it is always skipped when the default partition is a foreign table.

Attaching a partition acquires a SHARE UPDATE EXCLUSIVE lock on the parent table, in addition to the ACCESS EXCLUSIVE locks on the table being attached and on the default partition (if any).

Further locks must also be held on all sub−partitions if the table being attached is itself a partitioned table. Likewise if the default partition is itself a partitioned table. The locking of the sub−partitions can be avoided by adding a CHECK constraint as described in Section 5.12.2.2.

DETACH PARTITION *partition_name* [CONCURRENTLY | FINALIZE]

This form detaches the specified partition of the target table. The detached partition continues to exist as a standalone table, but no longer has any ties to the table from which it was detached. Any indexes that were attached to the target table's indexes are detached. Any triggers that were created as clones of those in the target table are removed. SHARE lock is obtained on any tables that reference this partitioned table in foreign key constraints.

If CONCURRENTLY is specified, it runs using a reduced lock level to avoid blocking other sessions that might be accessing the partitioned table. In this mode, two transactions are used internally. During the first transaction, a SHARE UPDATE EXCLUSIVE lock is taken on both parent table and partition, and the partition is marked as undergoing detach; at that point, the transaction is committed and all other transactions using the partitioned table are waited for. Once all those transactions have completed, the second transaction acquires SHARE UPDATE EXCLUSIVE on the partitioned table and ACCESS EXCLUSIVE on the partition, and the detach process completes. A CHECK constraint that duplicates the partition constraint is added to the partition. CONCURRENTLY cannot be run in a transaction block and is not allowed if the partitioned table contains a default partition.

If FINALIZE is specified, a previous DETACH CONCURRENTLY invocation that was canceled or

interrupted is completed. At most one partition in a partitioned table can be pending detach at a time.

All the forms of **ALTER TABLE** that act on a single table, except RENAME, SET SCHEMA, ATTACH PARTITION, and DETACH PARTITION can be combined into a list of multiple alterations to be applied together. For example, it is possible to add several columns and/or alter the type of several columns in a single command. This is particularly useful with large tables, since only one pass over the table need be made.

You must own the table to use **ALTER TABLE**. To change the schema or tablespace of a table, you must also have CREATE privilege on the new schema or tablespace. To add the table as a new child of a parent table, you must own the parent table as well. Also, to attach a table as a new partition of the table, you must own the table being attached. To alter the owner, you must be able to SET ROLE to the new owning role, and that role must have CREATE privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.) To add a column or alter a column type or use the OF clause, you must also have USAGE privilege on the data type.

PARAMETERS

IF EXISTS
: Do not throw an error if the table does not exist. A notice is issued in this case.

name
: The name (optionally schema−qualified) of an existing table to alter. If ONLY is specified before the table name, only that table is altered. If ONLY is not specified, the table and all its descendant tables (if any) are altered. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

column_name
: Name of a new or existing column.

new_column_name
: New name for an existing column.

new_name
: New name for the table.

data_type
: Data type of the new column, or new data type for an existing column.

table_constraint
: New table constraint for the table.

constraint_name
: Name of a new or existing constraint.

CASCADE
: Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column), and in turn all objects that depend on those objects (see Section 5.15).

RESTRICT
: Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

trigger_name
: Name of a single trigger to disable or enable.

ALL
: Disable or enable all triggers belonging to the table. (This requires superuser privilege if any of the triggers are internally generated constraint triggers, such as those that are used to implement foreign key constraints or deferrable uniqueness and exclusion constraints.)

USER
: Disable or enable all triggers belonging to the table except for internally generated constraint triggers,

such as those that are used to implement foreign key constraints or deferrable uniqueness and exclusion constraints.

index_name
The name of an existing index.

storage_parameter
The name of a table storage parameter.

value
The new value for a table storage parameter. This might be a number or a word depending on the parameter.

parent_table
A parent table to associate or de−associate with this table.

new_owner
The user name of the new owner of the table.

new_access_method
The name of the access method to which the table will be converted.

new_tablespace
The name of the tablespace to which the table will be moved.

new_schema
The name of the schema to which the table will be moved.

partition_name
The name of the table to attach as a new partition or to detach from this table.

partition_bound_spec
The partition bound specification for a new partition. Refer to CREATE TABLE (**CREATE_TABLE**(7)) for more details on the syntax of the same.

NOTES

The key word COLUMN is noise and can be omitted.

When a column is added with ADD COLUMN and a non−volatile DEFAULT is specified, the default value is evaluated at the time of the statement and the result stored in the table's metadata, where it will be returned when any existing rows are accessed. The value will be only applied when the table is rewritten, making the **ALTER TABLE** very fast even on large tables. If no column constraints are specified, NULL is used as the DEFAULT. In neither case is a rewrite of the table required.

Adding a column with a volatile DEFAULT (e.g., **clock_timestamp()**), a stored generated column, an identity column, or a column with a domain data type that has constraints will cause the entire table and its indexes to be rewritten. Adding a virtual generated column never requires a rewrite.

Changing the type of an existing column will normally cause the entire table and its indexes to be rewritten. As an exception, when changing the type of an existing column, if the USING clause does not change the column contents and the old type is either binary coercible to the new type or an unconstrained domain over the new type, a table rewrite is not needed. However, indexes will still be rebuilt unless the system can verify that the new index would be logically equivalent to the existing one. For example, if the collation for a column has been changed, an index rebuild is required because the new sort order might be different. However, in the absence of a collation change, a column can be changed from text to varchar (or vice versa) without rebuilding the indexes because these data types sort identically.

Table and/or index rebuilds may take a significant amount of time for a large table, and will temporarily require as much as double the disk space.

Adding a CHECK or NOT NULL constraint requires scanning the table to verify that existing rows meet the constraint, but does not require a table rewrite. If a CHECK constraint is added as NOT ENFORCED, no verification will be performed.

Similarly, when attaching a new partition it may be scanned to verify that existing rows meet the partition

constraint.

The main reason for providing the option to specify multiple changes in a single **ALTER TABLE** is that multiple table scans or rewrites can thereby be combined into a single pass over the table.

Scanning a large table to verify new foreign−key, check, or not−null constraints can take a long time, and other updates to the table are locked out until the **ALTER TABLE ADD CONSTRAINT** command is committed. The main purpose of the NOT VALID constraint option is to reduce the impact of adding a constraint on concurrent updates. With NOT VALID, the **ADD CONSTRAINT** command does not scan the table and can be committed immediately. After that, a VALIDATE CONSTRAINT command can be issued to verify that existing rows satisfy the constraint. The validation step does not need to lock out concurrent updates, since it knows that other transactions will be enforcing the constraint for rows that they insert or update; only pre−existing rows need to be checked. Hence, validation acquires only a SHARE UPDATE EXCLUSIVE lock on the table being altered. (If the constraint is a foreign key then a ROW SHARE lock is also required on the table referenced by the constraint.) In addition to improving concurrency, it can be useful to use NOT VALID and VALIDATE CONSTRAINT in cases where the table is known to contain pre−existing violations. Once the constraint is in place, no new violations can be inserted, and the existing problems can be corrected at leisure until VALIDATE CONSTRAINT finally succeeds.

The DROP COLUMN form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on−disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

To force immediate reclamation of space occupied by a dropped column, you can execute one of the forms of **ALTER TABLE** that performs a rewrite of the whole table. This results in reconstructing each row with the dropped column replaced by a null value.

The rewriting forms of **ALTER TABLE** are not MVCC−safe. After a table rewrite, the table will appear empty to concurrent transactions, if they are using a snapshot taken before the rewrite occurred. See Section 13.6 for more details.

The USING option of SET DATA TYPE can actually specify any expression involving the old values of the row; that is, it can refer to other columns as well as the one being converted. This allows very general conversions to be done with the SET DATA TYPE syntax. Because of this flexibility, the USING expression is not applied to the column's default value (if any); the result might not be a constant expression as required for a default. This means that when there is no implicit or assignment cast from old to new type, SET DATA TYPE might fail to convert the default even though a USING clause is supplied. In such cases, drop the default with DROP DEFAULT, perform the ALTER TYPE, and then use SET DEFAULT to add a suitable new default. Similar considerations apply to indexes and constraints involving the column.

If a table has any descendant tables, it is not permitted to add, rename, or change the type of a column in the parent table without doing the same to the descendants. This ensures that the descendants always have columns matching the parent. Similarly, a CHECK constraint cannot be renamed in the parent without also renaming it in all descendants, so that CHECK constraints also match between the parent and its descendants. (That restriction does not apply to index−based constraints, however.) Also, because selecting from the parent also selects from its descendants, a constraint on the parent cannot be marked valid unless it is also marked valid for those descendants. In all of these cases, **ALTER TABLE ONLY** will be rejected.

A recursive DROP COLUMN operation will remove a descendant table's column only if the descendant does not inherit that column from any other parents and never had an independent definition of the column. A nonrecursive DROP COLUMN (i.e., **ALTER TABLE ONLY ... DROP COLUMN**) never removes any descendant columns, but instead marks them as independently defined rather than inherited. A nonrecursive DROP COLUMN command will fail for a partitioned table, because all partitions of a table must have the same columns as the partitioning root.

The actions for identity columns (ADD GENERATED, SET etc., DROP IDENTITY), as well as the actions CLUSTER, OWNER, and TABLESPACE never recurse to descendant tables; that is, they always act as

though ONLY were specified. Actions affecting trigger states recurse to partitions of partitioned tables (unless ONLY is specified), but never to traditional−inheritance descendants. Adding a constraint recurses only for CHECK constraints that are not marked NO INHERIT.

Changing any part of a system catalog table is not permitted.

Refer to CREATE TABLE (**CREATE_TABLE**(7)) for a further description of valid parameters. Chapter 5 has further information on inheritance.

EXAMPLES

To add a column of type varchar to a table:

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

That will cause all existing rows in the table to be filled with null values for the new column.

To add a column with a non−null default:

```
ALTER TABLE measurements
  ADD COLUMN mtime timestamp with time zone DEFAULT now();
```

Existing rows will be filled with the current time as the value of the new column, and then new rows will receive the time of their insertion.

To add a column and fill it with a value different from the default to be used later:

```
ALTER TABLE transactions
  ADD COLUMN status varchar(30) DEFAULT 'old',
  ALTER COLUMN status SET default 'current';
```

Existing rows will be filled with old, but then the default for subsequent commands will be current. The effects are the same as if the two sub−commands had been issued in separate **ALTER TABLE** commands.

To drop a column from a table:

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

To change the types of two existing columns in one operation:

```
ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

To change an integer column containing Unix timestamps to timestamp with time zone via a USING clause:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

The same, when the column has a default expression that won't automatically cast to the new data type:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

To rename an existing constraint:

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

To add a not−null constraint to a column:

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

To remove a not−null constraint from a column:

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

To add a check constraint to a table and all its children:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

To add a check constraint only to a table and not to its children:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO INHERIT;
```

(The check constraint will not be inherited by future children, either.)

To remove a check constraint from a table and all its children:

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

To remove a check constraint from one table only:

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

(The check constraint remains in place for any child tables.)

To add a foreign key constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (address);
```

To add a foreign key constraint to a table with the least impact on other work:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (address) NOT
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

To add a (multicolumn) unique constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

To add an automatically named primary key constraint to a table, noting that a table can only ever have one primary key:

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

To move a table to a different tablespace:

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

To move a table to a different schema:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

To recreate a primary key constraint, without blocking updates while the index is rebuilt:

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
  ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

To attach a partition to a range−partitioned table:

```
ALTER TABLE measurement
  ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM ('2016−07−01') TO ('2016−08−01');
```

To attach a partition to a list−partitioned table:

```
ALTER TABLE cities
  ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

To attach a partition to a hash−partitioned table:

```
ALTER TABLE orders
  ATTACH PARTITION orders_p4 FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

To attach a default partition to a partitioned table:

```
ALTER TABLE cities
  ATTACH PARTITION cities_partdef DEFAULT;
```

To detach a partition from a partitioned table:

```
ALTER TABLE measurement
  DETACH PARTITION measurement_y2015m12;
```

COMPATIBILITY

The forms ADD [COLUMN], DROP [COLUMN], DROP IDENTITY, RESTART, SET DEFAULT, SET DATA TYPE (without USING), SET GENERATED, and SET *sequence_option* conform with the SQL standard. The form ADD *table_constraint* conforms with the SQL standard when the USING INDEX and NOT VALID clauses are omitted and the constraint type is one of CHECK, UNIQUE, PRIMARY KEY, or REFERENCES. The other forms are PostgreSQL extensions of the SQL standard. Also, the ability to specify more than one manipulation in a single **ALTER TABLE** command is an extension.

ALTER TABLE DROP COLUMN can be used to drop the only column of a table, leaving a zero−column table. This is an extension of SQL, which disallows zero−column tables.

SEE ALSO

CREATE TABLE (**CREATE_TABLE**(7))