ALTER TABLE

Name

https://www.postgresql.org/docs/current/static/sql-altertable.html#AEN67134

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 ]

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 STATISTICS integer
    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 }
    ADD table_constraint [ NOT VALID ]
    ADD table_constraint_using_index
    ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY
DEFERRED | INITIALLY IMMEDIATE ]
    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 WITH OIDS
    SET WITHOUT OIDS
    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_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }

and table_constraint_using_index is:

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

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 held unless explicitly noted. When multiple subcommands are listed, the lock held will be the strictest one required from 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. 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.

`SET/DROP DEFAULT`

These forms set or remove the default value for a column. Default values only apply in subsequent `INSERT` or `UPDATE` commands; they do 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. You can only use `SET NOT NULL` when the column contains no null values.

`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; alternatively, set it to -1 to revert to using the system default statistics target (default_statistics_target). 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. When set to a positive value, `ANALYZE` will assume that the column contains exactly the specified number of distinct nonnull values. When set to a negative value, which must be greater than or equal to -1, `ANALYZE` will assume that the number of distinct nonnull values in the column is linear in the size of the table; the exact count is to be computed by multiplying the estimated table size by the absolute value of the given number. 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 the average. This can be useful when the size of the table changes over time, since the multiplication by the number of rows in the table is not performed until query planning time. Specify a value of 0 to revert to estimating the number of distinct values normally.

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

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. 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 SET STORAGE doesn't itself change anything in the table, it just sets the strategy to be pursued during future table updates. See Section 65.2 for more information.

ADD table_constraint [NOT VALID]

This form adds a new constraint to a table using the same syntax as CREATE TABLE, plus the option NOT VALID, which is currently only allowed for foreign key and CHECK constraints. If the constraint is marked NOT VALID, the potentially-lengthy initial check to verify that all rows in the table satisfy the constraint is skipped. The constraint will still be enforced 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; and they'll fail unless the new row matches the specified check constraints). 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.

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.

> **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 INDEX CONCURRENTLY`, and then install it as an official 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.

`VALIDATE CONSTRAINT`

This form validates a foreign key or check 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. Nothing happens if the constraint is already marked valid.

Validation can be a long process on larger tables. The value of separating validation from initial creation is that you can defer validation to less busy times, or can be used to give additional time to correct pre-existing errors while preventing new errors. Note also that validation on its own does not prevent normal write commands against the table while it runs.

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.

`DROP CONSTRAINT [ IF EXISTS ]`

This form drops the specified constraint on a table. 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 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 mode.

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.

`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 WITH OIDS`

This form adds an `oid` system column to the table (see Section 5.4). It does nothing if the table already has OIDs.

Note that this is not equivalent to `ADD COLUMN oid oid`; that would add a normal column that happened to be named `oid`, not a system column.

`SET WITHOUT OIDS`

This form removes the `oid` system column from the table. This is exactly equivalent to `DROP COLUMN oid RESTRICT`, except that it will not complain if there is already no `oid` column.

`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. 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.

`SET ( storage_parameter = value [, ... ] )`

This form changes one or more storage parameters for the table. See Storage Parameters 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.

Changing fillfactor and autovacuum storage parameters acquires a `SHARE UPDATE EXCLUSIVE` lock.

> **Note:** While `CREATE TABLE` allows `OIDS` to be specified in the `WITH (storage_parameter)` syntax, `ALTER TABLE` does not treat `OIDS` as a storage parameter. Instead use the `SET WITH OIDS` and `SET WITHOUT OIDS` forms to change OID status.

`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, and if they have `NOT NULL` constraints in the parent then they must also have `NOT NULL` constraints in the child.

There must also be matching child-table constraints for all `CHECK` constraints of the parent, except those marked non-inheritable (that is, created with `ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT`) in the parent, 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 presence of an `oid` system column is permitted to differ. 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`

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. This option has no effect except when logical replication is in use. `DEFAULT` (the default for non-system tables) records the old values of the columns of the primary key, if any. `USING INDEX` records the old values of the columns covered by the named index, which must be unique, not partial, not deferrable, and include only columns marked `NOT NULL`. `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.) In all cases, no old values are logged unless at least one of the columns that would be logged differs between the old and new versions of the row.

`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. 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.

All the forms of ALTER TABLE that act on a single table, except `RENAME`, and `SET SCHEMA` can be combined into a list of multiple alterations to 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. To alter the owner, you must also be a direct or indirect member of 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 5.13).

`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_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.

Notes

The key word `COLUMN` is noise and can be omitted.

When a column is added with `ADD COLUMN`, all existing rows in the table are initialized with the column's default value (NULL if no `DEFAULT` clause is specified). If there is no `DEFAULT` clause, this is merely a metadata change and does not require any immediate update of the table's data; the added NULL values are supplied on readout, instead.

Adding a column with a `DEFAULT` clause or changing the type of an existing column will require 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; but any indexes on the affected columns must still be rebuilt. Adding or removing a system `oid` column also requires rewriting the entire table. 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.

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.

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. (These statements do not apply when dropping the system `oid` column; that is done with an immediate rewrite.)

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 13.5 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, or rename an inherited constraint in the parent table without doing the same to the descendants. That is, `ALTER TABLE ONLY` will be rejected. This ensures that the descendants always have columns matching the parent.

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.

The `TRIGGER`, `CLUSTER`, `OWNER`, and `TABLESPACE` actions never recurse to descendant tables; that is, they always act as though `ONLY` were specified. 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 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);
```

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 VALID;
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;
```

Compatibility

The forms `ADD` (without `USING INDEX`), `DROP`, `SET DEFAULT`, and `SET DATA TYPE` (without `USING`) conform with the SQL standard. 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
