NAME

CREATE_INDEX − define a new index

SYNOPSIS

CREATE [UNIQUE] INDEX [CONCURRENTLY] [[IF NOT EXISTS] *name*] ON [ONLY] *table_name* [USING *meth*
({ *column_name* | (*expression*) } [COLLATE *collation*] [*opclass* [(*opclass_parameter* = *value* [, ...])]] [ASC | DESC
[INCLUDE (*column_name* [, ...])]
[NULLS [NOT] DISTINCT]
[WITH (*storage_parameter* [= *value*] [, ...])]
[TABLESPACE *tablespace_name*]
[WHERE *predicate*]

DESCRIPTION

CREATE INDEX constructs an index on the specified column(s) of the specified relation, which can be a table or a materialized view. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on upper(col) would allow the clause WHERE upper(col) = 'JIM' to use an index.

PostgreSQL provides the index methods B−tree, hash, GiST, SP−GiST, GIN, and BRIN. Users can also define their own index methods, but that is fairly complicated.

When the WHERE clause is present, a partial index is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use WHERE with UNIQUE to enforce uniqueness over a subset of a table. See Section 11.8 for more discussion.

The expression used in the WHERE clause can refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subqueries and aggregate expressions are also forbidden in WHERE. The same restrictions apply to index fields that are expressions.

All functions and operators used in an index definition must be “immutable”, that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well−defined. To use a user−defined function in an index expression or WHERE clause, remember to mark the function immutable when you create it.

PARAMETERS

UNIQUE

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

Additional restrictions apply when unique indexes are applied to partitioned tables; see CREATE TABLE (**CREATE_TABLE**(7)).

CONCURRENTLY

When this option is used, PostgreSQL will build the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table; whereas a standard index build locks out writes (but not reads) on the table until it's done. There are several caveats to be aware of when using this option — see Building Indexes Concurrently below.

For temporary tables, **CREATE INDEX** is always non−concurrent, as no other session can access them, and non−concurrent index creation is cheaper.

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 index is anything like the one that would have been created. Index name is required when IF NOT EXISTS is specified.

INCLUDE
The optional INCLUDE clause specifies a list of columns which will be included in the index as non−key columns. A non−key column cannot be used in an index scan search qualification, and it is disregarded for purposes of any uniqueness or exclusion constraint enforced by the index. However, an index−only scan can return the contents of non−key columns without having to visit the index's table, since they are available directly from the index entry. Thus, addition of non−key columns allows index−only scans to be used for queries that otherwise could not use them.

It's wise to be conservative about adding non−key columns to an index, especially wide columns. If an index tuple exceeds the maximum size allowed for the index type, data insertion will fail. In any case, non−key columns duplicate data from the index's table and bloat the size of the index, thus potentially slowing searches. Furthermore, B−tree deduplication is never used with indexes that have a non−key column.

Columns listed in the INCLUDE clause don't need appropriate operator classes; the clause can include columns whose data types don't have operator classes defined for a given access method.

Expressions are not supported as included columns since they cannot be used in index−only scans.

Currently, the B−tree, GiST and SP−GiST index access methods support this feature. In these indexes, the values of columns listed in the INCLUDE clause are included in leaf tuples which correspond to heap tuples, but are not included in upper−level index entries used for tree navigation.

name
The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table. The name of the index must be distinct from the name of any other relation (table, sequence, index, view, materialized view, or foreign table) in that schema. If the name is omitted, PostgreSQL chooses a suitable name based on the parent table's name and the indexed column name(s).

ONLY
Indicates not to recurse creating indexes on partitions, if the table is partitioned. The default is to recurse.

table_name
The name (possibly schema−qualified) of the table to be indexed.

method
The name of the index method to be used. Choices are btree, hash, gist, spgist, gin, brin, or user−installed access methods like bloom. The default method is btree.

column_name
The name of a column of the table.

expression
An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses can be omitted if the expression has the form of a function call.

collation
The name of the collation to use for the index. By default, the index uses the collation declared for the column to be indexed or the result collation of the expression to be indexed. Indexes with non−default

collations can be useful for queries that involve expressions using non−default collations.

opclass
The name of an operator class. See below for details.

opclass_parameter
The name of an operator class parameter. See below for details.

ASC
Specifies ascending sort order (which is the default).

DESC
Specifies descending sort order.

NULLS FIRST
Specifies that nulls sort before non−nulls. This is the default when DESC is specified.

NULLS LAST
Specifies that nulls sort after non−nulls. This is the default when DESC is not specified.

NULLS DISTINCT
NULLS NOT DISTINCT
Specifies whether for a unique index, null values should be considered distinct (not equal). The default is that they are distinct, so that a unique index could contain multiple null values in a column.

storage_parameter
The name of an index−method−specific storage parameter. See Index Storage Parameters below for details.

tablespace_name
The tablespace in which to create the index. If not specified, default_tablespace is consulted, or temp_tablespaces for indexes on temporary tables.

predicate
The constraint expression for a partial index.

Index Storage Parameters

The optional WITH clause specifies storage parameters for the index. Each index method has its own set of allowed storage parameters.

The B−tree, hash, GiST and SP−GiST index methods all accept this parameter:

fillfactor (integer)
Controls how full the index method will try to pack index pages. For B−trees, leaf pages are filled to this percentage during initial index builds, and also when extending the index at the right (adding new largest key values). If pages subsequently become completely full, they will be split, leading to fragmentation of the on−disk index structure. B−trees use a default fillfactor of 90, but any integer value from 10 to 100 can be selected.

B−tree indexes on tables where many inserts and/or updates are anticipated can benefit from lower fillfactor settings at **CREATE INDEX** time (following bulk loading into the table). Values in the range of 50 − 90 can usefully “smooth out” the *rate* of page splits during the early life of the B−tree index (lowering fillfactor like this may even lower the absolute number of page splits, though this effect is highly workload dependent). The B−tree bottom−up index deletion technique described in Section 65.1.4.2 is dependent on having some “extra” space on pages to store “extra” tuple versions, and so can be affected by fillfactor (though the effect is usually not significant).

In other specific cases it might be useful to increase fillfactor to 100 at **CREATE INDEX** time as a way of maximizing space utilization. You should only consider this when you are completely sure that the table is static (i.e. that it will never be affected by either inserts or updates). A fillfactor setting of 100 otherwise risks *harming* performance: even a few updates or inserts will cause a sudden flood of page splits.

The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

B−tree indexes additionally accept this parameter:

deduplicate_items (boolean)
Controls usage of the B−tree deduplication technique described in Section 65.1.4.3. Set to ON or OFF to enable or disable the optimization. (Alternative spellings of ON and OFF are allowed as described in Section 19.1.) The default is ON.

Note
Turning deduplicate_items off via **ALTER INDEX** prevents future insertions from triggering deduplication, but does not in itself make existing posting list tuples use the standard tuple representation.

GiST indexes additionally accept this parameter:

buffering (enum)
Controls whether the buffered build technique described in Section 65.2.4.1 is used to build the index. With OFF buffering is disabled, with ON it is enabled, and with AUTO it is initially disabled, but is turned on on−the−fly once the index size reaches effective_cache_size. The default is AUTO. Note that if sorted build is possible, it will be used instead of buffered build unless buffering=ON is specified.

GIN indexes accept these parameters:

fastupdate (boolean)
Controls usage of the fast update technique described in Section 65.4.4.1. ON enables fast update, OFF disables it. The default is ON.

Note
Turning fastupdate off via **ALTER INDEX** prevents future insertions from going into the list of pending index entries, but does not in itself flush existing entries. You might want to **VACUUM** the table or call the **gin_clean_pending_list** function afterward to ensure the pending list is emptied.

gin_pending_list_limit (integer)
Overrides the global setting of gin_pending_list_limit for this index. This value is specified in kilobytes.

BRIN indexes accept these parameters:

pages_per_range (integer)
Defines the number of table blocks that make up one block range for each entry of a BRIN index (see Section 65.5.1 for more details). The default is 128.

autosummarize (boolean)
Defines whether a summarization run is queued for the previous page range whenever an insertion is detected on the next one (see Section 65.5.1.1 for more details). The default is off.

Building Indexes Concurrently

Creating an index can interfere with regular operation of a database. Normally PostgreSQL locks the table to be indexed against writes and performs the entire index build with a single scan of the table. Other transactions can still read the table, but if they try to insert, update, or delete rows in the table they will block until the index build is finished. This could have a severe effect if the system is a live production database. Very large tables can take many hours to be indexed, and even for smaller tables, an index build can lock out writers for periods that are unacceptably long for a production system.

PostgreSQL supports building indexes without locking out writes. This method is invoked by specifying the CONCURRENTLY option of **CREATE INDEX**. When this option is used, PostgreSQL must perform two scans of the table, and in addition it must wait for all existing transactions that could potentially modify or use the index to terminate. Thus this method requires more total work than a standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is

built, this method is useful for adding new indexes in a production environment. Of course, the extra CPU and I/O load imposed by the index creation might slow other operations.

In a concurrent index build, the index is actually entered as an “invalid” index into the system catalogs in one transaction, then two table scans occur in two more transactions. Before each table scan, the index build must wait for existing transactions that have modified the table to terminate. After the second scan, the index build must wait for any transactions that have a snapshot (see Chapter 13) predating the second scan to terminate, including transactions used by any phase of concurrent index builds on other tables, if the indexes involved are partial or have columns that are not simple column references. Then finally the index can be marked “valid” and ready for use, and the**CREA TE INDEX** command terminates. Even then, however, the index may not be immediately usable for queries: in the worst case, it cannot be used as long as transactions exist that predate the start of the index build.

If a problem arises while scanning the table, such as a deadlock or a uniqueness violation in a unique index, the **CREATE INDEX** command will fail but leave behind an “invalid” index. This index will be ignored for querying purposes because it might be incomplete; however it will still consume update overhead. The psql **\d** command will report such an index as INVALID:

```
postgres=# \d tab
      Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
Indexes:
   "idx" btree (col) INVALID
```

The recommended recovery method in such cases is to drop the index and try again to perform **CREATE INDEX CONCURRENTLY**. (Another possibility is to rebuild the index with **REINDEX INDEX CONCURRENTLY**).

Another caveat when building a unique index concurrently is that the uniqueness constraint is already being enforced against other transactions when the second table scan begins. This means that constraint violations could be reported in other queries prior to the index becoming available for use, or even in cases where the index build eventually fails. Also, if a failure does occur in the second scan, the “invalid” index continues to enforce its uniqueness constraint afterwards.

Concurrent builds of expression indexes and partial indexes are supported. Errors occurring in the evaluation of these expressions could cause behavior similar to that described above for unique constraint violations.

Regular index builds permit other regular index builds on the same table to occur simultaneously, but only one concurrent index build can occur on a table at a time. In either case, schema modification of the table is not allowed while the index is being built. Another difference is that a regular **CREATE INDEX** command can be performed within a transaction block, but **CREATE INDEX CONCURRENTLY** cannot.

Concurrent builds for indexes on partitioned tables are currently not supported. However, you may concurrently build the index on each partition individually and then finally create the partitioned index non−concurrently in order to reduce the time where writes to the partitioned table will be locked out. In this case, building the partitioned index is a metadata only operation.

NOTES

See Chapter 11 for information about when indexes can be used, when they are not used, and in which particular situations they can be useful.

Currently, only the B−tree, GiST, GIN, and BRIN index methods support multiple−key−column indexes. Whether there can be multiple key columns is independent of whether INCLUDE columns can be added to the index. Indexes can have up to 32 columns, including INCLUDE columns. (This limit can be altered when building PostgreSQL.) Only B−tree currently supports unique indexes.

An operator class with optional parameters can be specified for each column of an index. The operator class

identifies the operators to be used by the index for that column. For example, a B−tree index on four−byte integers would use the int4_ops class; this operator class includes comparison functions for four−byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex−number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when creating an index. More information about operator classes is in Section 11.10 and in Section 36.16.

When CREATE INDEX is invoked on a partitioned table, the default behavior is to recurse to all partitions to ensure they all have matching indexes. Each partition is first checked to determine whether an equivalent index already exists, and if so, that index will become attached as a partition index to the index being created, which will become its parent index. If no matching index exists, a new index will be created and automatically attached; the name of the new index in each partition will be determined as if no index name had been specified in the command. If the ONLY option is specified, no recursion is done, and the index is marked invalid. (**ALTER INDEX ... ATTACH PARTITION** marks the index valid, once all partitions acquire matching indexes.) Note, however, that any partition that is created in the future using **CREATE TABLE ... PARTITION OF** will automatically have a matching index, regardless of whether ONLY is specified.

For index methods that support ordered scans (currently, only B−tree), the optional clauses ASC, DESC, NULLS FIRST, and/or NULLS LAST can be specified to modify the sort ordering of the index. Since an ordered index can be scanned either forward or backward, it is not normally useful to create a single−column DESC index — that sort ordering is already available with a regular index. The value of these options is that multicolumn indexes can be created that match the sort ordering requested by a mixed−ordering query, such as SELECT ... ORDER BY x ASC, y DESC. The NULLS options are useful if you need to support “nulls sort low” behavior, rather than the default “nulls sort high”, in queries that depend on indexes to avoid sorting steps.

The system regularly collects statistics on all of a table's columns. Newly−created non−expression indexes can immediately use these statistics to determine an index's usefulness. For new expression indexes, it is necessary to run **ANALYZE** or wait for the autovacuum daemon to analyze the table to generate statistics for these indexes.

While **CREATE INDEX** is running, the search_path is temporarily changed to pg_catalog, pg_temp.

For most index methods, the speed of creating an index is dependent on the setting of maintenance_work_mem. Larger values will reduce the time needed for index creation, so long as you don't make it larger than the amount of memory really available, which would drive the machine into swapping.

PostgreSQL can build indexes while leveraging multiple CPUs in order to process the table rows faster. This feature is known as parallel index build. For index methods that support building indexes in parallel (currently, B−tree, GIN, and BRIN), *maintenance_work_mem* specifies the maximum amount of memory that can be used by each index build operation as a whole, regardless of how many worker processes were started. Generally, a cost model automatically determines how many worker processes should be requested, if any.

Parallel index builds may benefit from increasing *maintenance_work_mem* where an equivalent serial index build will see little or no benefit. Note that *maintenance_work_mem* may influence the number of worker processes requested, since parallel workers must have at least a 32MB share of the total *maintenance_work_mem* budget. There must also be a remaining 32MB share for the leader process. Increasing max_parallel_maintenance_workers may allow more workers to be used, which will reduce the time needed for index creation, so long as the index build is not already I/O bound. Of course, there should also be sufficient CPU capacity that would otherwise lie idle.

Setting a value for parallel_workers via **ALTER TABLE** directly controls how many parallel worker processes will be requested by a **CREATE INDEX** against the table. This bypasses the cost model completely, and prevents *maintenance_work_mem* from affecting how many parallel workers are requested. Setting parallel_workers to 0 via **ALTER TABLE** will disable parallel index builds on the table in all

cases.

Tip

You might want to reset parallel_workers after setting it as part of tuning an index build. This avoids inadvertent changes to query plans, since parallel_workers affects *all* parallel table scans.

While **CREATE INDEX** with the CONCURRENTLY option supports parallel builds without special restrictions, only the first table scan is actually performed in parallel.

Use **DROP INDEX** to remove an index.

Like any long−running transaction, **CREATE INDEX** on a table can affect which tuples can be removed by concurrent **VACUUM** on any other table.

Prior releases of PostgreSQL also had an R−tree index method. This method has been removed because it had no significant advantages over the GiST method. If USING rtree is specified, **CREATE INDEX** will interpret it as USING gist, to simplify conversion of old databases to GiST.

Each backend running **CREATE INDEX** will report its progress in the pg_stat_progress_create_index view. See Section 27.4.4 for details.

EXAMPLES

To create a unique B−tree index on the column title in the table films:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

To create a unique B−tree index on the column title with included columns director and rating in the table films:

```
CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);
```

To create a B−Tree index with deduplication disabled:

```
CREATE INDEX title_idx ON films (title) WITH (deduplicate_items = off);
```

To create an index on the expression lower(title), allowing efficient case−insensitive searches:

```
CREATE INDEX ON films ((lower(title)));
```

(In this example we have chosen to omit the index name, so the system will choose a name, typically films_lower_idx.)

To create an index with non−default collation:

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

To create an index with non−default sort ordering of nulls:

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

To create an index with non−default fill factor:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

To create a GIN index with fast updates disabled:

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);
```

To create an index on the column code in the table films and have the index reside in the tablespace indexspace:

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

To create a GiST index on a point attribute so that we can efficiently use box operators on the result of the

conversion function:

```
CREATE INDEX pointloc
   ON points USING gist (box(location,location));
SELECT * FROM points
   WHERE box(location,location) && '(0,0),(1,1)'::box;
```

To create an index without locking out writes to the table:

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

COMPATIBILITY

CREATE INDEX is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

SEE ALSO

ALTER INDEX (**ALTER_INDEX**(7)), DROP INDEX (**DROP_INDEX**(7)), **REINDEX**(7), Section 27.4.4