NAME

CREATE_TYPE − define a new data type

SYNOPSIS

```
CREATE TYPE name AS
  ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
  ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
  SUBTYPE = subtype
  [ , SUBTYPE_OPCLASS = subtype_operator_class ]
  [ , COLLATION = collation ]
  [ , CANONICAL = canonical_function ]
  [ , SUBTYPE_DIFF = subtype_diff_function ]
  [ , MULTIRANGE_TYPE_NAME = multirange_type_name ]
)

CREATE TYPE name (
  INPUT = input_function,
  OUTPUT = output_function
  [ , RECEIVE = receive_function ]
  [ , SEND = send_function ]
  [ , TYPMOD_IN = type_modifier_input_function ]
  [ , TYPMOD_OUT = type_modifier_output_function ]
  [ , ANALYZE = analyze_function ]
  [ , SUBSCRIPT = subscript_function ]
  [ , INTERNALLENGTH = { internallength | VARIABLE } ]
  [ , PASSEDBYVALUE ]
  [ , ALIGNMENT = alignment ]
  [ , STORAGE = storage ]
  [ , LIKE = like_type ]
  [ , CATEGORY = category ]
  [ , PREFERRED = preferred ]
  [ , DEFAULT = default ]
  [ , ELEMENT = element ]
  [ , DELIMITER = delimiter ]
  [ , COLLATABLE = collatable ]
)

CREATE TYPE name
```

DESCRIPTION

CREATE TYPE registers a new data type for use in the current database. The user who defines a type becomes its owner.

If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. (Because tables have associated data types, the type name must also be distinct from the name of any existing table in the same schema.)

There are five forms of **CREATE TYPE**, as shown in the syntax synopsis above. They respectively create a composite type, an enum type, a range type, a base type, or a shell type. The first four of these are discussed in turn below. A shell type is simply a placeholder for a type to be defined later; it is created by issuing **CREATE TYPE** with no parameters except for the type name. Shell types are needed as forward references when creating range types and base types, as discussed in those sections.

Composite Types

The first form of **CREATE TYPE** creates a composite type. The composite type is specified by a list of attribute names and data types. An attribute's collation can be specified too, if its data type is collatable. A composite type is essentially the same as the row type of a table, but using **CREATE TYPE** avoids the need to create an actual table when all that is wanted is to define a type. A stand−alone composite type is useful, for example, as the argument or return type of a function.

To be able to create a composite type, you must have USAGE privilege on all attribute types.

Enumerated Types

The second form of **CREATE TYPE** creates an enumerated (enum) type, as described in Section 8.7. Enum types take a list of quoted labels, each of which must be less than NAMEDATALEN bytes long (64 bytes in a standard PostgreSQL build). (It is possible to create an enumerated type with zero labels, but such a type cannot be used to hold values before at least one label is added using **ALTER TYPE**.)

Range Types

The third form of **CREATE TYPE** creates a new range type, as described in Section 8.17.

The range type's *subtype* can be any type with an associated b−tree operator class (to determine the ordering of values for the range type). Normally the subtype's default b−tree operator class is used to determine ordering; to use a non−default operator class, specify its name with *subtype_opclass*. If the subtype is collatable, and you want to use a non−default collation in the range's ordering, specify the desired collation with the *collation* option.

The optional *canonical* function must take one argument of the range type being defined, and return a value of the same type. This is used to convert range values to a canonical form, when applicable. See Section 8.17.8 for more information. Creating a *canonical* function is a bit tricky, since it must be defined before the range type can be declared. To do this, you must first create a shell type, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command CREATE TYPE *name*, with no additional parameters. Then the function can be declared using the shell type as argument and result, and finally the range type can be declared using the same name. This automatically replaces the shell type entry with a valid range type.

The optional *subtype_diff* function must take two values of the *subtype* type as argument, and return a double precision value representing the difference between the two given values. While this is optional, providing it allows much greater efficiency of GiST indexes on columns of the range type. See Section 8.17.8 for more information.

The optional *multirange_type_name* parameter specifies the name of the corresponding multirange type. If not specified, this name is chosen automatically as follows. If the range type name contains the substring range, then the multirange type name is formed by replacement of the range substring with multirange in the range type name. Otherwise, the multirange type name is formed by appending a _multirange suffix to the range type name.

Base Types

The fourth form of **CREATE TYPE** creates a new base type (scalar type). To create a new base type, you must be a superuser. (This restriction is made because an erroneous type definition could confuse or even crash the server.)

The parameters can appear in any order, not only that illustrated above, and most are optional. You must register two or more functions (using **CREATE FUNCTION**) before defining the type. The support functions *input_function* and *output_function* are required, while the functions *receive_function*, *send_function*, *type_modifier_input_function*, *type_modifier_output_function*, *analyze_function*, and *subscript_function* are optional. Generally these functions have to be coded in C or another low−level language.

The *input_function* converts the type's external textual representation to the internal representation used by the operators and functions defined for the type. *output_function* performs the reverse transformation. The input function can be declared as taking one argument of type cstring, or as taking three arguments of types cstring, oid, integer. The first argument is the input text as a C string, the second argument is the type's own

OID (except for array types, which instead receive their element type's OID), and the third is the typmod of the destination column, if known (−1 will be passed if not). The input function must return a value of the data type itself. Usually, an input function should be declared STRICT; if it is not, it will be called with a NULL first parameter when reading a NULL input value. The function must still return NULL in this case, unless it raises an error. (This case is mainly meant to support domain input functions, which might need to reject NULL inputs.) The output function must be declared as taking one argument of the new data type. The output function must return type cstring. Output functions are not invoked for NULL values.

The optional *receive_function* converts the type's external binary representation to the internal representation. If this function is not supplied, the type cannot participate in binary input. The binary representation should be chosen to be cheap to convert to internal form, while being reasonably portable. (For example, the standard integer data types use network byte order as the external binary representation, while the internal representation is in the machine's native byte order.) The receive function should perform adequate checking to ensure that the value is valid. The receive function can be declared as taking one argument of type internal, or as taking three arguments of types internal, oid, integer. The first argument is a pointer to a StringInfo buffer holding the received byte string; the optional arguments are the same as for the text input function. The receive function must return a value of the data type itself. Usually, a receive function should be declared STRICT; if it is not, it will be called with a NULL first parameter when reading a NULL input value. The function must still return NULL in this case, unless it raises an error. (This case is mainly meant to support domain receive functions, which might need to reject NULL inputs.) Similarly, the optional *send_function* converts from the internal representation to the external binary representation. If this function is not supplied, the type cannot participate in binary output. The send function must be declared as taking one argument of the new data type. The send function must return type bytea. Send functions are not invoked for NULL values.

You should at this point be wondering how the input and output functions can be declared to have results or arguments of the new type, when they have to be created before the new type can be created. The answer is that the type should first be defined as a shell type, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command CREATE TYPE *name*, with no additional parameters. Then the C I/O functions can be defined referencing the shell type. Finally, **CREATE TYPE** with a full definition replaces the shell entry with a complete, valid type definition, after which the new type can be used normally.

The optional *type_modifier_input_function* and *type_modifier_output_function* are needed if the type supports modifiers, that is optional constraints attached to a type declaration, such as char(5) or numeric(30,2). PostgreSQL allows user−defined types to take one or more simple constants or identifiers as modifiers. However, this information must be capable of being packed into a single non−negative integer value for storage in the system catalogs. The *type_modifier_input_function* is passed the declared modifier(s) in the form of a cstring array. It must check the values for validity (throwing an error if they are wrong), and if they are correct, return a single non−negative integer value that will be stored as the column “typmod”. Type modifiers will be rejected if the type does not have a *type_modifier_input_function*. The *type_modifier_output_function* converts the internal integer typmod value back to the correct form for user display. It must return a cstring value that is the exact string to append to the type name; for example numeric's function might return (30,2). It is allowed to omit the *type_modifier_output_function*, in which case the default display format is just the stored typmod integer value enclosed in parentheses.

The optional *analyze_function* performs type−specific statistics collection for columns of the data type. By default, **ANALYZE** will attempt to gather statistics using the type's “equals” and “less−than” operators, if there is a default b−tree operator class for the type. For non−scalar types this behavior is likely to be unsuitable, so it can be overridden by specifying a custom analysis function. The analysis function must be declared to take a single argument of type internal, and return a boolean result. The detailed API for analysis functions appears in src/include/commands/vacuum.h.

The optional *subscript_function* allows the data type to be subscripted in SQL commands. Specifying this function does not cause the type to be considered a “true” array type; for example, it will not be a candidate for the result type of ARRAY[] constructs. But if subscripting a value of the type is a natural notation for extracting data from it, then a *subscript_function* can be written to define what that means. The subscript

function must be declared to take a single argument of type internal, and return an internal result, which is a pointer to a struct of methods (functions) that implement subscripting. The detailed API for subscript functions appears in src/include/nodes/subscripting.h. It may also be useful to read the array implementation in src/backend/utils/adt/arraysubs.c, or the simpler code in contrib/hstore/hstore_subs.c. Additional information appears in Array Types below.

While the details of the new type's internal representation are only known to the I/O functions and other functions you create to work with the type, there are several properties of the internal representation that must be declared to PostgreSQL. Foremost of these is *internallength*. Base data types can be fixed−length, in which case *internallength* is a positive integer, or variable−length, indicated by setting *internallength* to VARIABLE. (Internally, this is represented by setting typlen to −1.) The internal representation of all variable−length types must start with a 4−byte integer giving the total length of this value of the type. (Note that the length field is often encoded, as described in Section 66.2; it's unwise to access it directly.)

The optional flag PASSEDBYVALUE indicates that values of this data type are passed by value, rather than by reference. Types passed by value must be fixed−length, and their internal representation cannot be larger than the size of the Datum type (4 bytes on some machines, 8 bytes on others).

The *alignment* parameter specifies the storage alignment required for the data type. The allowed values equate to alignment on 1, 2, 4, or 8 byte boundaries. Note that variable−length types must have an alignment of at least 4, since they necessarily contain an int4 as their first component.

The *storage* parameter allows selection of storage strategies for variable−length data types. (Only plain is allowed for fixed−length types.) plain specifies that data of the type will always be stored in−line and not compressed. extended specifies that the system will first try to compress a long data value, and will move the value out of the main table row if it's still too long. external allows the value to be moved out of the main table, but the system will not try to compress it. main allows compression, but discourages moving the value out of the main table. (Data items with this storage strategy might still be moved out of the main table if there is no other way to make a row fit, but they will be kept in the main table preferentially over extended and external items.)

All *storage* values other than plain imply that the functions of the data type can handle values that have been toasted, as described in Section 66.2 and Section 36.13.1. The specific other value given merely determines the default TOAST storage strategy for columns of a toastable data type; users can pick other strategies for individual columns using ALTER TABLE SET STORAGE.

The *like_type* parameter provides an alternative method for specifying the basic representation properties of a data type: copy them from some existing type. The values of *internallength*, *passedbyvalue*, *alignment*, and *storage* are copied from the named type. (It is possible, though usually undesirable, to override some of these values by specifying them along with the LIKE clause.) Specifying representation this way is especially useful when the low−level implementation of the new type “piggybacks” on an existing type in some fashion.

The *category* and *preferred* parameters can be used to help control which implicit cast will be applied in ambiguous situations. Each data type belongs to a category named by a single ASCII character, and each type is either “preferred” or not within its category. The parser will prefer casting to preferred types (but only from other types within the same category) when this rule is helpful in resolving overloaded functions or operators. For more details see Chapter 10. For types that have no implicit casts to or from any other types, it is sufficient to leave these settings at the defaults. However, for a group of related types that have implicit casts, it is often helpful to mark them all as belonging to a category and select one or two of the “most general” types as being preferred within the category. The *category* parameter is especially useful when adding a user−defined type to an existing built−in category, such as the numeric or string types. However, it is also possible to create new entirely−user−defined type categories. Select any ASCII character other than an upper−case letter to name such a category.

A default value can be specified, in case a user wants columns of the data type to default to something other than the null value. Specify the default with the DEFAULT key word. (Such a default can be overridden by an explicit DEFAULT clause attached to a particular column.)

To indicate that a type is a fixed−length array type, specify the type of the array elements using the

ELEMENT key word. For example, to define an array of 4−byte integers (int4), specify ELEMENT = int4. For more details, see Array Types below.

To indicate the delimiter to be used between values in the external representation of arrays of this type, *delimiter* can be set to a specific character. The default delimiter is the comma (,). Note that the delimiter is associated with the array element type, not the array type itself.

If the optional Boolean parameter *collatable* is true, column definitions and expressions of the type may carry collation information through use of the COLLATE clause. It is up to the implementations of the functions operating on the type to actually make use of the collation information; this does not happen automatically merely by marking the type collatable.

Array Types

Whenever a user−defined type is created, PostgreSQL automatically creates an associated array type, whose name consists of the element type's name prepended with an underscore, and truncated if necessary to keep it less than NAMEDATALEN bytes long. (If the name so generated collides with an existing type name, the process is repeated until a non−colliding name is found.) This implicitly−created array type is variable length and uses the built−in input and output functions array_in and array_out. Furthermore, this type is what the system uses for constructs such as ARRAY[] over the user−defined type. The array type tracks any changes in its element type's owner or schema, and is dropped if the element type is.

You might reasonably ask why there is an **ELEMENT** option, if the system makes the correct array type automatically. The main case where it's useful to use **ELEMENT** is when you are making a fixed−length type that happens to be internally an array of a number of identical things, and you want to allow these things to be accessed directly by subscripting, in addition to whatever operations you plan to provide for the type as a whole. For example, type point is represented as just two floating−point numbers, which can be accessed using point[0] and point[1]. Note that this facility only works for fixed−length types whose internal form is exactly a sequence of identical fixed−length fields. For historical reasons (i.e., this is clearly wrong but it's far too late to change it), subscripting of fixed−length array types starts from zero, rather than from one as for variable−length arrays.

Specifying the **SUBSCRIPT** option allows a data type to be subscripted, even though the system does not otherwise regard it as an array type. The behavior just described for fixed−length arrays is actually implemented by the **SUBSCRIPT** handler function **raw_array_subscript_handler**, which is used automatically if you specify **ELEMENT** for a fixed−length type without also writing **SUBSCRIPT**.

When specifying a custom **SUBSCRIPT** function, it is not necessary to specify **ELEMENT** unless the **SUBSCRIPT** handler function needs to consult typelem to find out what to return. Be aware that specifying **ELEMENT** causes the system to assume that the new type contains, or is somehow physically dependent on, the element type; thus for example changing properties of the element type won't be allowed if there are any columns of the dependent type.

PARAMETERS

name
: The name (optionally schema−qualified) of a type to be created.

attribute_name
: The name of an attribute (column) for the composite type.

data_type
: The name of an existing data type to become a column of the composite type.

collation
: The name of an existing collation to be associated with a column of a composite type, or with a range type.

label
: A string literal representing the textual label associated with one value of an enum type.

subtype
: The name of the element type that the range type will represent ranges of.

subtype_operator_class
The name of a b−tree operator class for the subtype.

canonical_function
The name of the canonicalization function for the range type.

subtype_diff_function
The name of a difference function for the subtype.

multirange_type_name
The name of the corresponding multirange type.

input_function
The name of a function that converts data from the type's external textual form to its internal form.

output_function
The name of a function that converts data from the type's internal form to its external textual form.

receive_function
The name of a function that converts data from the type's external binary form to its internal form.

send_function
The name of a function that converts data from the type's internal form to its external binary form.

type_modifier_input_function
The name of a function that converts an array of modifier(s) for the type into internal form.

type_modifier_output_function
The name of a function that converts the internal form of the type's modifier(s) to external textual form.

analyze_function
The name of a function that performs statistical analysis for the data type.

subscript_function
The name of a function that defines what subscripting a value of the data type does.

internallength
A numeric constant that specifies the length in bytes of the new type's internal representation. The default assumption is that it is variable−length.

alignment
The storage alignment requirement of the data type. If specified, it must be char, int2, int4, or double; the default is int4.

storage
The storage strategy for the data type. If specified, must be plain, external, extended, or main; the default is plain.

like_type
The name of an existing data type that the new type will have the same representation as. The values of *internallength*, *passedbyvalue*, *alignment*, and *storage* are copied from that type, unless overridden by explicit specification elsewhere in this **CREATE TYPE** command.

category
The category code (a single ASCII character) for this type. The default is 'U' for “user−defined type”. Other standard category codes can be found in Table 52.65. You may also choose other ASCII characters in order to create custom categories.

preferred
True if this type is a preferred type within its type category, else false. The default is false. Be very careful about creating a new preferred type within an existing type category, as this could cause surprising changes in behavior.

default

The default value for the data type. If this is omitted, the default is null.

element
The type being created is an array; this specifies the type of the array elements.

delimiter
The delimiter character to be used between values in arrays made of this type.

collatable
True if this type's operations can use collation information. The default is false.

NOTES

Because there are no restrictions on use of a data type once it's been created, creating a base type or range type is tantamount to granting public execute permission on the functions mentioned in the type definition. This is usually not an issue for the sorts of functions that are useful in a type definition. But you might want to think twice before designing a type in a way that would require “secret” information to be used while converting it to or from external form.

Before PostgreSQL version 8.3, the name of a generated array type was always exactly the element type's name with one underscore character (_) prepended. (Type names were therefore restricted in length to one fewer character than other names.) While this is still usually the case, the array type name may vary from this in case of maximum−length names or collisions with user type names that begin with underscore. Writing code that depends on this convention is therefore deprecated. Instead, use pg_type.typarray to locate the array type associated with a given type.

It may be advisable to avoid using type and table names that begin with underscore. While the server will change generated array type names to avoid collisions with user−given names, there is still risk of confusion, particularly with old client software that may assume that type names beginning with underscores always represent arrays.

Before PostgreSQL version 8.2, the shell−type creation syntax CREATE TYPE *name* did not exist. The way to create a new base type was to create its input function first. In this approach, PostgreSQL will first see the name of the new data type as the return type of the input function. The shell type is implicitly created in this situation, and then it can be referenced in the definitions of the remaining I/O functions. This approach still works, but is deprecated and might be disallowed in some future release. Also, to avoid accidentally cluttering the catalogs with shell types as a result of simple typos in function definitions, a shell type will only be made this way when the input function is written in C.

In PostgreSQL version 16 and later, it is desirable for base types' input functions to return “soft” errors using the new **errsave()**/**ereturn()** mechanism, rather than throwing **ereport()** exceptions as in previous versions. See src/backend/utils/fmgr/README for more information.

EXAMPLES

This example creates a composite type and uses it in a function definition:

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
   SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

This example creates an enumerated type and uses it in a table definition:

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
   id serial,
   description text,
   status bug_status
);
```

This example creates a range type:

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

This example creates the base data type box and then uses the type in a table definition:

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
   INTERNALLENGTH = 16,
   INPUT = my_box_in_function,
   OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
   id integer,
   description box
);
```

If the internal structure of box were an array of four float4 elements, we might instead use:

```
CREATE TYPE box (
   INTERNALLENGTH = 16,
   INPUT = my_box_in_function,
   OUTPUT = my_box_out_function,
   ELEMENT = float4
);
```

which would allow a box value's component numbers to be accessed by subscripting. Otherwise the type behaves the same as before.

This example creates a large object type and uses it in a table definition:

```
CREATE TYPE bigobj (
   INPUT = lo_filein, OUTPUT = lo_fileout,
   INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
   id integer,
   obj bigobj
);
```

More examples, including suitable input and output functions, are in Section 36.13.

COMPATIBILITY

The first form of the **CREATE TYPE** command, which creates a composite type, conforms to the SQL standard. The other forms are PostgreSQL extensions. The **CREATE TYPE** statement in the SQL standard also defines other forms that are not implemented in PostgreSQL.

The ability to create a composite type with zero attributes is a PostgreSQL−specific deviation from the standard (analogous to the same case in **CREATE TABLE**).

SEE ALSO

ALTER TYPE (**ALTER_TYPE**(7)), CREATE DOMAIN (**CREATE_DOMAIN**(7)), CREATE FUNCTION (**CREATE_FUNCTION**(7)), DROP TYPE (**DROP_TYPE**(7))