**8.14. Arrays**

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in or user-defined base type, enum type, or composite type can be created. Arrays of domains are not yet supported.

**8.14.1. Declaration of Array Types**

To illustrate the use of array types, we create this table:

CREATE TABLE sal\_emp (

name text,

pay\_by\_quarter integer[],

schedule text[][]

);

As shown, an array data type is named by appending square brackets ([]) to the data type name of the array elements. The above command will create a table named sal\_emp with a column of type text (name), a one-dimensional array of type integer (pay\_by\_quarter), which represents the employee's salary by quarter, and a two-dimensional array of text (schedule), which represents the employee's weekly schedule.

The syntax for CREATE TABLE allows the exact size of arrays to be specified, for example:

CREATE TABLE tictactoe (

squares integer[3][3]

);

However, the current implementation ignores any supplied array size limits, i.e., the behavior is the same as for arrays of unspecified length.

The current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring the array size or number of dimensions in CREATE TABLE is simply documentation; it does not affect run-time behavior.

An alternative syntax, which conforms to the SQL standard by using the keyword ARRAY, can be used for one-dimensional arrays. pay\_by\_quarter could have been defined as:

pay\_by\_quarter integer ARRAY[4],

Or, if no array size is to be specified:

pay\_by\_quarter integer ARRAY,

As before, however, PostgreSQL does not enforce the size restriction in any case.

**8.14.2. Array Value Input**

To write an array value as a literal constant, enclose the element values within curly braces and separate them by commas. (If you know C, this is not unlike the C syntax for initializing structures.) You can put double quotes around any element value, and must do so if it contains commas or curly braces. (More details appear below.) Thus, the general format of an array constant is the following:

'{ ***val1*** ***delim*** ***val2*** ***delim*** ... }'

where ***delim*** is the delimiter character for the type, as recorded in its pg\_type entry. Among the standard data types provided in the PostgreSQL distribution, all use a comma (,), except for type box which uses a semicolon (;). Each ***val*** is either a constant of the array element type, or a subarray. An example of an array constant is:

'{{1,2,3},{4,5,6},{7,8,9}}'

This constant is a two-dimensional, 3-by-3 array consisting of three subarrays of integers.

To set an element of an array constant to NULL, write NULL for the element value. (Any upper- or lower-case variant of NULL will do.) If you want an actual string value "NULL", you must put double quotes around it.

(These kinds of array constants are actually only a special case of the generic type constants discussed in [**Section 4.1.2.7**](https://www.postgresql.org/docs/9.1/sql-syntax-lexical.html#SQL-SYNTAX-CONSTANTS-GENERIC). The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

Now we can show some INSERT statements:

INSERT INTO sal\_emp

VALUES ('Bill',

'{10000, 10000, 10000, 10000}',

'{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal\_emp

VALUES ('Carol',

'{20000, 25000, 25000, 25000}',

'{{"breakfast", "consulting"}, {"meeting", "lunch"}}');

The result of the previous two inserts looks like this:

SELECT \* FROM sal\_emp;

name | pay\_by\_quarter | schedule

-------+---------------------------+-------------------------------------------

Bill | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}

Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}

(2 rows)

Multidimensional arrays must have matching extents for each dimension. A mismatch causes an error, for example:

INSERT INTO sal\_emp

VALUES ('Bill',

'{10000, 10000, 10000, 10000}',

'{{"meeting", "lunch"}, {"meeting"}}');

ERROR: multidimensional arrays must have array expressions with matching dimensions

The ARRAY constructor syntax can also be used:

INSERT INTO sal\_emp

VALUES ('Bill',

ARRAY[10000, 10000, 10000, 10000],

ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal\_emp

VALUES ('Carol',

ARRAY[20000, 25000, 25000, 25000],

ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);

Notice that the array elements are ordinary SQL constants or expressions; for instance, string literals are single quoted, instead of double quoted as they would be in an array literal. The ARRAY constructor syntax is discussed in more detail in [**Section 4.2.12**](https://www.postgresql.org/docs/9.1/sql-expressions.html#SQL-SYNTAX-ARRAY-CONSTRUCTORS).

**8.14.3. Accessing Arrays**

Now, we can run some queries on the table. First, we show how to access a single element of an array. This query retrieves the names of the employees whose pay changed in the second quarter:

SELECT name FROM sal\_emp WHERE pay\_by\_quarter[1] <> pay\_by\_quarter[2];

name

-------

Carol

(1 row)

The array subscript numbers are written within square brackets. By default PostgreSQL uses a one-based numbering convention for arrays, that is, an array of ***n*** elements starts with array[1] and ends with array[***n***].

This query retrieves the third quarter pay of all employees:

SELECT pay\_by\_quarter[3] FROM sal\_emp;

pay\_by\_quarter

----------------

10000

25000

(2 rows)

We can also access arbitrary rectangular slices of an array, or subarrays. An array slice is denoted by writing ***lower-bound***:***upper-bound*** for one or more array dimensions. For example, this query retrieves the first item on Bill's schedule for the first two days of the week:

SELECT schedule[1:2][1:1] FROM sal\_emp WHERE name = 'Bill';

schedule

------------------------

{{meeting},{training}}

(1 row)

If any dimension is written as a slice, i.e., contains a colon, then all dimensions are treated as slices. Any dimension that has only a single number (no colon) is treated as being from 1 to the number specified. For example, [2] is treated as [1:2], as in this example:

SELECT schedule[1:2][2] FROM sal\_emp WHERE name = 'Bill';

schedule

-------------------------------------------

{{meeting,lunch},{training,presentation}}

(1 row)

To avoid confusion with the non-slice case, it's best to use slice syntax for all dimensions, e.g., [1:2][1:1], not [2][1:1].

An array subscript expression will return null if either the array itself or any of the subscript expressions are null. Also, null is returned if a subscript is outside the array bounds (this case does not raise an error). For example, if schedule currently has the dimensions [1:3][1:2] then referencing schedule[3][3] yields NULL. Similarly, an array reference with the wrong number of subscripts yields a null rather than an error.

An array slice expression likewise yields null if the array itself or any of the subscript expressions are null. However, in other cases such as selecting an array slice that is completely outside the current array bounds, a slice expression yields an empty (zero-dimensional) array instead of null. (This does not match non-slice behavior and is done for historical reasons.) If the requested slice partially overlaps the array bounds, then it is silently reduced to just the overlapping region instead of returning null.

The current dimensions of any array value can be retrieved with the array\_dims function:

SELECT array\_dims(schedule) FROM sal\_emp WHERE name = 'Carol';

array\_dims

------------

[1:2][1:2]

(1 row)

array\_dims produces a text result, which is convenient for people to read but perhaps inconvenient for programs. Dimensions can also be retrieved with array\_upper and array\_lower, which return the upper and lower bound of a specified array dimension, respectively:

SELECT array\_upper(schedule, 1) FROM sal\_emp WHERE name = 'Carol';

array\_upper

-------------

2

(1 row)

array\_length will return the length of a specified array dimension:

SELECT array\_length(schedule, 1) FROM sal\_emp WHERE name = 'Carol';

array\_length

--------------

2

(1 row)

**8.14.4. Modifying Arrays**

An array value can be replaced completely:

UPDATE sal\_emp SET pay\_by\_quarter = '{25000,25000,27000,27000}'

WHERE name = 'Carol';

or using the ARRAY expression syntax:

UPDATE sal\_emp SET pay\_by\_quarter = ARRAY[25000,25000,27000,27000]

WHERE name = 'Carol';

An array can also be updated at a single element:

UPDATE sal\_emp SET pay\_by\_quarter[4] = 15000

WHERE name = 'Bill';

or updated in a slice:

UPDATE sal\_emp SET pay\_by\_quarter[1:2] = '{27000,27000}'

WHERE name = 'Carol';

A stored array value can be enlarged by assigning to elements not already present. Any positions between those previously present and the newly assigned elements will be filled with nulls. For example, if array myarray currently has 4 elements, it will have six elements after an update that assigns to myarray[6]; myarray[5] will contain null. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

Subscripted assignment allows creation of arrays that do not use one-based subscripts. For example one might assign to myarray[-2:7] to create an array with subscript values from -2 to 7.

New array values can also be constructed using the concatenation operator, ||:

SELECT ARRAY[1,2] || ARRAY[3,4];

?column?

-----------

{1,2,3,4}

(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];

?column?

---------------------

{{5,6},{1,2},{3,4}}

(1 row)

The concatenation operator allows a single element to be pushed onto the beginning or end of a one-dimensional array. It also accepts two ***N***-dimensional arrays, or an ***N***-dimensional and an ***N+1***-dimensional array.

When a single element is pushed onto either the beginning or end of a one-dimensional array, the result is an array with the same lower bound subscript as the array operand. For example:

SELECT array\_dims(1 || '[0:1]={2,3}'::int[]);

array\_dims

------------

[0:2]

(1 row)

SELECT array\_dims(ARRAY[1,2] || 3);

array\_dims

------------

[1:3]

(1 row)

When two arrays with an equal number of dimensions are concatenated, the result retains the lower bound subscript of the left-hand operand's outer dimension. The result is an array comprising every element of the left-hand operand followed by every element of the right-hand operand. For example:

SELECT array\_dims(ARRAY[1,2] || ARRAY[3,4,5]);

array\_dims

------------

[1:5]

(1 row)

SELECT array\_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);

array\_dims

------------

[1:5][1:2]

(1 row)

When an ***N***-dimensional array is pushed onto the beginning or end of an ***N+1***-dimensional array, the result is analogous to the element-array case above. Each ***N***-dimensional sub-array is essentially an element of the ***N+1***-dimensional array's outer dimension. For example:

SELECT array\_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);

array\_dims

------------

[1:3][1:2]

(1 row)

An array can also be constructed by using the functions array\_prepend, array\_append, or array\_cat. The first two only support one-dimensional arrays, but array\_cat supports multidimensional arrays. Note that the concatenation operator discussed above is preferred over direct use of these functions. In fact, these functions primarily exist for use in implementing the concatenation operator. However, they might be directly useful in the creation of user-defined aggregates. Some examples:

SELECT array\_prepend(1, ARRAY[2,3]);

array\_prepend

---------------

{1,2,3}

(1 row)

SELECT array\_append(ARRAY[1,2], 3);

array\_append

--------------

{1,2,3}

(1 row)

SELECT array\_cat(ARRAY[1,2], ARRAY[3,4]);

array\_cat

-----------

{1,2,3,4}

(1 row)

SELECT array\_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);

array\_cat

---------------------

{{1,2},{3,4},{5,6}}

(1 row)

SELECT array\_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);

array\_cat

---------------------

{{5,6},{1,2},{3,4}}

**8.14.5. Searching in Arrays**

To search for a value in an array, each value must be checked. This can be done manually, if you know the size of the array. For example:

SELECT \* FROM sal\_emp WHERE pay\_by\_quarter[1] = 10000 OR

pay\_by\_quarter[2] = 10000 OR

pay\_by\_quarter[3] = 10000 OR

pay\_by\_quarter[4] = 10000;

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is unknown. An alternative method is described in [**Section 9.21**](https://www.postgresql.org/docs/9.1/functions-comparisons.html). The above query could be replaced by:

SELECT \* FROM sal\_emp WHERE 10000 = ANY (pay\_by\_quarter);

In addition, you can find rows where the array has all values equal to 10000 with:

SELECT \* FROM sal\_emp WHERE 10000 = ALL (pay\_by\_quarter);

Alternatively, the generate\_subscripts function can be used. For example:

SELECT \* FROM

(SELECT pay\_by\_quarter,

generate\_subscripts(pay\_by\_quarter, 1) AS s

FROM sal\_emp) AS foo

WHERE pay\_by\_quarter[s] = 10000;

This function is described in [**Table 9-47**](https://www.postgresql.org/docs/9.1/functions-srf.html#FUNCTIONS-SRF-SUBSCRIPTS).

**Tip:** Arrays are not sets; searching for specific array elements can be a sign of database misdesign. Consider using a separate table with a row for each item that would be an array element. This will be easier to search, and is likely to scale better for a large number of elements.

**8.14.6. Array Input and Output Syntax**

The external text representation of an array value consists of items that are interpreted according to the I/O conversion rules for the array's element type, plus decoration that indicates the array structure. The decoration consists of curly braces ({ and }) around the array value plus delimiter characters between adjacent items. The delimiter character is usually a comma (,) but can be something else: it is determined by the typdelim setting for the array's element type. Among the standard data types provided in the PostgreSQL distribution, all use a comma, except for type box, which uses a semicolon (;). In a multidimensional array, each dimension (row, plane, cube, etc.) gets its own level of curly braces, and delimiters must be written between adjacent curly-braced entities of the same level.

The array output routine will put double quotes around element values if they are empty strings, contain curly braces, delimiter characters, double quotes, backslashes, or white space, or match the word NULL. Double quotes and backslashes embedded in element values will be backslash-escaped. For numeric data types it is safe to assume that double quotes will never appear, but for textual data types one should be prepared to cope with either the presence or absence of quotes.

By default, the lower bound index value of an array's dimensions is set to one. To represent arrays with other lower bounds, the array subscript ranges can be specified explicitly before writing the array contents. This decoration consists of square brackets ([]) around each array dimension's lower and upper bounds, with a colon (:) delimiter character in between. The array dimension decoration is followed by an equal sign (=). For example:

SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2

FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

e1 | e2

----+----

1 | 6

(1 row)

The array output routine will include explicit dimensions in its result only when there are one or more lower bounds different from one.

If the value written for an element is NULL (in any case variant), the element is taken to be NULL. The presence of any quotes or backslashes disables this and allows the literal string value "NULL" to be entered. Also, for backward compatibility with pre-8.2 versions of PostgreSQL, the **[array\_nulls](https://www.postgresql.org/docs/9.1/runtime-config-compatible.html" \l "GUC-ARRAY-NULLS)** configuration parameter can be turned off to suppress recognition of NULL as a NULL.

As shown previously, when writing an array value you can use double quotes around any individual array element. You must do so if the element value would otherwise confuse the array-value parser. For example, elements containing curly braces, commas (or the data type's delimiter character), double quotes, backslashes, or leading or trailing whitespace must be double-quoted. Empty strings and strings matching the word NULL must be quoted, too. To put a double quote or backslash in a quoted array element value, use escape string syntax and precede it with a backslash. Alternatively, you can avoid quotes and use backslash-escaping to protect all data characters that would otherwise be taken as array syntax.

You can add whitespace before a left brace or after a right brace. You can also add whitespace before or after any individual item string. In all of these cases the whitespace will be ignored. However, whitespace within double-quoted elements, or surrounded on both sides by non-whitespace characters of an element, is not ignored.

**9.18. Array Functions and Operators**

[**Table 9-47**](https://www.postgresql.org/docs/9.6/functions-array.html#ARRAY-OPERATORS-TABLE) shows the operators available for array types.

**Table 9-47. Array Operators**

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| = | equal | ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3] | t |
| <> | not equal | ARRAY[1,2,3] <> ARRAY[1,2,4] | t |
| < | less than | ARRAY[1,2,3] < ARRAY[1,2,4] | t |
| > | greater than | ARRAY[1,4,3] > ARRAY[1,2,4] | t |
| <= | less than or equal | ARRAY[1,2,3] <= ARRAY[1,2,3] | t |
| >= | greater than or equal | ARRAY[1,4,3] >= ARRAY[1,4,3] | t |
| @> | contains | ARRAY[1,4,3] @> ARRAY[3,1,3] | t |
| <@ | is contained by | ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6] | t |
| && | overlap (have elements in common) | ARRAY[1,4,3] && ARRAY[2,1] | t |
| || | array-to-array concatenation | ARRAY[1,2,3] || ARRAY[4,5,6] | {1,2,3,4,5,6} |
| || | array-to-array concatenation | ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]] | {{1,2,3},{4,5,6},{7,8,9}} |
| || | element-to-array concatenation | 3 || ARRAY[4,5,6] | {3,4,5,6} |
| || | array-to-element concatenation | ARRAY[4,5,6] || 7 | {4,5,6,7} |

The array ordering operators (<, >=, etc) compare the array contents element-by-element, using the default B-tree comparison function for the element data type, and sort based on the first difference. In multidimensional arrays the elements are visited in row-major order (last subscript varies most rapidly). If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order. (This is a change from versions of PostgreSQL prior to 8.2: older versions would claim that two arrays with the same contents were equal, even if the number of dimensions or subscript ranges were different.)

The array containment operators (<@ and @>) consider one array to be contained in another one if each of its elements appears in the other one. Duplicates are not treated specially, thus ARRAY[1] and ARRAY[1,1] are each considered to contain the other.

See [**Section 8.15**](https://www.postgresql.org/docs/9.6/arrays.html) for more details about array operator behavior. See [**Section 11.2**](https://www.postgresql.org/docs/9.6/indexes-types.html) for more details about which operators support indexed operations.

[**Table 9-48**](https://www.postgresql.org/docs/9.6/functions-array.html#ARRAY-FUNCTIONS-TABLE) shows the functions available for use with array types. See [**Section 8.15**](https://www.postgresql.org/docs/9.6/arrays.html) for more information and examples of the use of these functions.

**Table 9-48. Array Functions**

| **Function** | **Return Type** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| array\_append(anyarray, anyelement) | anyarray | append an element to the end of an array | array\_append(ARRAY[1,2], 3) | {1,2,3} |
| array\_cat(anyarray, anyarray) | anyarray | concatenate two arrays | array\_cat(ARRAY[1,2,3], ARRAY[4,5]) | {1,2,3,4,5} |
| array\_ndims(anyarray) | int | returns the number of dimensions of the array | array\_ndims(ARRAY[[1,2,3], [4,5,6]]) | 2 |
| array\_dims(anyarray) | text | returns a text representation of array's dimensions | array\_dims(ARRAY[[1,2,3], [4,5,6]]) | [1:2][1:3] |
| array\_fill(anyelement, int[] [, int[]]) | anyarray | returns an array initialized with supplied value and dimensions, optionally with lower bounds other than 1 | array\_fill(7, ARRAY[3], ARRAY[2]) | [2:4]={7,7,7} |
| array\_length(anyarray, int) | int | returns the length of the requested array dimension | array\_length(array[1,2,3], 1) | 3 |
| array\_lower(anyarray, int) | int | returns lower bound of the requested array dimension | array\_lower('[0:2]={1,2,3}'::int[], 1) | 0 |
| array\_position(anyarray, anyelement [, int]) | int | returns the subscript of the first occurrence of the second argument in the array, starting at the element indicated by the third argument or at the first element (array must be one-dimensional) | array\_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon') | 2 |
| array\_positions(anyarray, anyelement) | int[] | returns an array of subscripts of all occurrences of the second argument in the array given as first argument (array must be one-dimensional) | array\_positions(ARRAY['A','A','B','A'], 'A') | {1,2,4} |
| array\_prepend(anyelement, anyarray) | anyarray | append an element to the beginning of an array | array\_prepend(1, ARRAY[2,3]) | {1,2,3} |
| array\_remove(anyarray, anyelement) | anyarray | remove all elements equal to the given value from the array (array must be one-dimensional) | array\_remove(ARRAY[1,2,3,2], 2) | {1,3} |
| array\_replace(anyarray, anyelement, anyelement) | anyarray | replace each array element equal to the given value with a new value | array\_replace(ARRAY[1,2,5,4], 5, 3) | {1,2,3,4} |
| array\_to\_string(anyarray, text [, text]) | text | concatenates array elements using supplied delimiter and optional null string | array\_to\_string(ARRAY[1, 2, 3, NULL, 5], ',', '\*') | 1,2,3,\*,5 |
| array\_upper(anyarray, int) | int | returns upper bound of the requested array dimension | array\_upper(ARRAY[1,8,3,7], 1) | 4 |
| cardinality(anyarray) | int | returns the total number of elements in the array, or 0 if the array is empty | cardinality(ARRAY[[1,2],[3,4]]) | 4 |
| string\_to\_array(text, text [, text]) | text[] | splits string into array elements using supplied delimiter and optional null string | string\_to\_array('xx~^~yy~^~zz', '~^~', 'yy') | {xx,NULL,zz} |
| unnest(anyarray) | setof anyelement | expand an array to a set of rows | unnest(ARRAY[1,2]) | 1  2  (2 rows) |
| unnest(anyarray, anyarray [, ...]) | setof anyelement, anyelement [, ...] | expand multiple arrays (possibly of different types) to a set of rows. This is only allowed in the FROM clause; see [**Section 7.2.1.4**](https://www.postgresql.org/docs/9.6/queries-table-expressions.html#QUERIES-TABLEFUNCTIONS) | unnest(ARRAY[1,2],ARRAY['foo','bar','baz']) | 1 foo  2 bar  NULL baz  (3 rows) |

In array\_position and array\_positions, each array element is compared to the searched value using IS NOT DISTINCT FROM semantics.

In array\_position, NULL is returned if the value is not found.

In array\_positions, NULL is returned only if the array is NULL; if the value is not found in the array, an empty array is returned instead.

In string\_to\_array, if the delimiter parameter is NULL, each character in the input string will become a separate element in the resulting array. If the delimiter is an empty string, then the entire input string is returned as a one-element array. Otherwise the input string is split at each occurrence of the delimiter string.

In string\_to\_array, if the null-string parameter is omitted or NULL, none of the substrings of the input will be replaced by NULL. In array\_to\_string, if the null-string parameter is omitted or NULL, any null elements in the array are simply skipped and not represented in the output string.