

create_aggregate



- [Description](#)
 - [Parameters](#)
 - [Notes](#)
 - [Examples](#)
 - [Compatibility](#)
 - [See Also](#)
-

CREATE AGGREGATE

CREATE AGGREGATE

7

SQL - Language Statements

CREATE AGGREGATE

define a new aggregate function

CREATE [OR REPLACE] AGGREGATE

name

([

argmode

][

argname

]

arg_data_type

[, ...] (SFUNC =

sfunc

, STYPE =

state_data_type

[, SSPACE =

state_data_size

][, FINALFUNC =

ffunc

][, FINALFUNC_EXTRA][, FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }][,
COMBINEFUNC =

combinefunc

][, SERIALFUNC =

serialfunc

][, DESERIALFUNC =

deserialfunc

][, INITCOND =

initial_condition

][, MSFUNC =

msfunc

][, MINVFUNC =

minvfunc

][, MSTYPE =

mstate_data_type

][, MSSPACE =

mstate_data_size

][, MFINALFUNC =

mffunc

[[, MFINALFUNC_EXTRA] [, MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }]
[, MINITCOND =

minitial_condition

] [, SORTOP =

sort_operator

] [, SUPPORT =

support_function

] [, PARALLEL = { SAFE | RESTRICTED | UNSAFE }]) CREATE [OR REPLACE] AGGREGATE

name

([[

argmode

] [

argname

]

arg_data_type

[, ...]] ORDER BY [

argmode

] [

argname

]

arg_data_type

[, ...]) (SFUNC =

sfunc

, STYPE =

state_data_type

[, SSPACE =

state_data_size

][, FINALFUNC =

ffunc

][, FINALFUNC_EXTRA][, FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }][,
INITCOND =

initial_condition

][, SUPPORT =

support_function

][, PARALLEL = { SAFE | RESTRICTED | UNSAFE }][, HYPOTHETICAL])

or the old syntax

CREATE [OR REPLACE] AGGREGATE

name

(BASETYPE =

base_type

, SFUNC =

sfunc

, STYPE =

state_data_type

[, SSPACE =

state_data_size

][, FINALFUNC =

ffunc

] [, FINALFUNC_EXTRA] [, FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }] [,
COMBINEFUNC =

combinefunc

] [, SERIALFUNC =

serialfunc

] [, DESERIALFUNC =

deserialfunc

] [, INITCOND =

initial_condition

] [, MSFUNC =

msfunc

] [, MINVFUNC =

minvfunc

] [, MSTYPE =

mstate_data_type

] [, MSSPACE =

mstate_data_size

] [, MFINALFUNC =

mffunc

] [, MFINALFUNC_EXTRA] [, MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }]
[, MINITCOND =

minitial_condition

] [, SORTOP =

sort_operator

] [, SUPPORT =

Description

`CREATE AGGREGATE` defines a new aggregate function. `CREATE OR REPLACE AGGREGATE` will either define a new aggregate function or replace an existing definition. Some basic and commonly-used aggregate functions are included with the distribution; they are documented in [???](#). If one defines new types or needs an aggregate function not already provided, then `CREATE AGGREGATE` can be used to provide the desired features.

When replacing an existing definition, the argument types, result type, and number of direct arguments may not be changed. Also, the new definition must be of the same kind (ordinary aggregate, ordered-set aggregate, or hypothetical-set aggregate) as the old one.

If a schema name is given (for example, `CREATE AGGREGATE myschema.myagg ...`) then the aggregate function is created in the specified schema. Otherwise it is created in the current schema.

An aggregate function is identified by its name and input data type(s). Two aggregates in the same schema can have the same name if they operate on different input types. The name and input data type(s) of an aggregate must also be distinct from the name and input data type(s) of every ordinary function in the same schema. This behavior is identical to overloading of ordinary function names (see [???](#)).

A simple aggregate function is made from one or two ordinary functions: a state transition function `<sfunc>`, and an optional final calculation function `<ffunc>`. These are used as follows:

```
sfunc( internal-state, next-data-values ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

PostgreSQL creates a temporary variable of data type `<stype>` to hold the current internal state of the aggregate. At each input row, the aggregate argument value(s) are calculated and the state transition function is invoked with the current state value and the new argument value(s) to calculate a new internal state value. After all the rows have been processed, the final function is invoked once to calculate the aggregate's return value. If there is no final function then the ending state value is returned as-is.

An aggregate function can provide an initial condition, that is, an initial value for the internal state value. This is specified and stored in the database as a value of type `text`, but it must be a valid external representation of a constant of the state value data type. If it is not supplied then the state value starts out null.

If the state transition function is declared “strict”, then it cannot be called with null inputs. With such a transition function, aggregate execution behaves as follows. Rows with any null input values are ignored (the function is not called and the previous state value is retained). If the initial state value is null, then at the first row with all-nonnull input values, the first argument value replaces the state value, and the transition function is invoked at each subsequent row with all-nonnull input values. This is handy for implementing aggregates like `max`. Note that this behavior is only available when `<state_data_type>` is the same as the first `<arg_data_type>`. When these types are different, you must supply a nonnull initial condition or use a nonstrict transition function.

If the state transition function is not strict, then it will be called unconditionally at each input row, and must deal with null inputs and null state values for itself. This allows the aggregate author to have full control over the aggregate's handling of null values.

If the final function is declared “strict”, then it will not be called when the ending state value is null; instead a null result will be returned automatically. (Of course this is just the normal behavior of strict functions.) In any case the final function has the option of returning a null value. For example, the final function for `avg` returns null when it sees there were zero input rows.

Sometimes it is useful to declare the final function as taking not just the state value, but extra parameters corresponding to the aggregate's input values. The main reason for doing this is if the final function is polymorphic and the state value's data type would be inadequate to pin down the result type. These extra parameters are always passed as NULL (and so the final function must not be strict when the `FINALFUNC_EXTRA` option is used), but nonetheless they are valid parameters. The final function could for example make use of `get_fn_expr_argtype` to identify the actual argument type in the current call.

An aggregate can optionally support moving-aggregate mode, as described in [???](#). This requires specifying the `MSFUNC`, `MINVFUNC`, and `MSTYPE` parameters, and optionally the `MSSPACE`, `MFINALFUNC`, `MFINALFUNC_EXTRA`, `MFINALFUNC_MODIFY`, and `MINITCOND` parameters. Except for `MINVFUNC`, these parameters work like the corresponding simple-aggregate parameters without `M`; they define a separate implementation of the aggregate that includes an inverse transition function.

The syntax with `ORDER BY` in the parameter list creates a special type of aggregate called an ordered-set aggregate; or if `HYPOTHETICAL` is specified, then a hypothetical-set aggregate is created. These aggregates operate over groups of sorted values in order-dependent ways, so that specification of an input sort order is an essential part of a call. Also, they can have direct arguments, which are arguments that are evaluated only once per aggregation rather than once per input row. Hypothetical-set aggregates are a subclass of ordered-set aggregates in which some of the direct arguments are required to match, in number and data types, the aggregated argument columns. This allows the values of those direct arguments to be added to the collection of aggregate-input rows as an additional “hypothetical” row.

An aggregate can optionally support partial aggregation, as described in [???](#). This requires specifying the `COMBINEFUNC` parameter. If the `<state_data_type>` is `internal`, it's usually also appropriate to provide the `SERIALFUNC` and `DESERIALFUNC` parameters so that parallel aggregation is

possible. Note that the aggregate must also be marked `PARALLEL SAFE` to enable parallel aggregation.

Aggregates that behave like `MIN` or `MAX` can sometimes be optimized by looking into an index instead of scanning every input row. If this aggregate can be so optimized, indicate it by specifying a sort operator. The basic requirement is that the aggregate must yield the first element in the sort ordering induced by the operator; in other words:

```
SELECT agg(col) FROM tab;
```

must be equivalent to:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Further assumptions are that the aggregate ignores null inputs, and that it delivers a null result if and only if there were no non-null inputs. Ordinarily, a data type's `<` operator is the proper sort operator for `MIN`, and `>` is the proper sort operator for `MAX`. Note that the optimization will never actually take effect unless the specified operator is the “less than” or “greater than” strategy member of a B-tree index operator class.

To be able to create an aggregate function, you must have `USAGE` privilege on the argument types, the state type(s), and the return type, as well as `EXECUTE` privilege on the supporting functions.

Parameters

`<name>`

The name (optionally schema-qualified) of the aggregate function to create.

`<argmode>`

The mode of an argument: `IN` or `VARIADIC`. (Aggregate functions do not support `OUT` arguments.) If omitted, the default is `IN`. Only the last argument can be marked `VARIADIC`.

`<argname>`

The name of an argument. This is currently only useful for documentation purposes. If omitted, the argument has no name.

`<arg_data_type>`

An input data type on which this aggregate function operates. To create a zero-argument aggregate function, write `*` in place of the list of argument specifications. (An example of such an aggregate is `count(*)`.)

`<base_type>`

In the old syntax for `CREATE AGGREGATE`, the input data type is specified by a `basetype` parameter

rather than being written next to the aggregate name. Note that this syntax allows only one input parameter. To define a zero-argument aggregate function with this syntax, specify the `basetype` as `"ANY"` (not `*`). Ordered-set aggregates cannot be defined with the old syntax.

`<sfunc>`

The name of the state transition function to be called for each input row. For a normal `<N>`-argument aggregate function, the `<sfunc>` must take `<N>+1` arguments, the first being of type `<state_data_type>` and the rest matching the declared input data type(s) of the aggregate. The function must return a value of type `<state_data_type>`. This function takes the current state value and the current input data value(s), and returns the next state value.

For ordered-set (including hypothetical-set) aggregates, the state transition function receives only the current state value and the aggregated arguments, not the direct arguments. Otherwise it is the same.

`<state_data_type>`

The data type for the aggregate's state value.

`<state_data_size>`

The approximate average size (in bytes) of the aggregate's state value. If this parameter is omitted or is zero, a default estimate is used based on the `<state_data_type>`. If set to a negative value, it indicates the state data can grow unboundedly in size, such as when the aggregate accumulates input rows (e.g., `array_agg`, `string_agg`). The planner uses this value to estimate the memory required for a grouped aggregate query and to avoid optimizations that may cause excessive memory usage.

`<ffunc>`

The name of the final function called to compute the aggregate's result after all input rows have been traversed. For a normal aggregate, this function must take a single argument of type `<state_data_type>`. The return data type of the aggregate is defined as the return type of this function. If `<ffunc>` is not specified, then the ending state value is used as the aggregate's result, and the return type is `<state_data_type>`.

For ordered-set (including hypothetical-set) aggregates, the final function receives not only the final state value, but also the values of all the direct arguments.

If `FINALFUNC_EXTRA` is specified, then in addition to the final state value and any direct arguments, the final function receives extra NULL values corresponding to the aggregate's regular (aggregated) arguments. This is mainly useful to allow correct resolution of the aggregate result type when a polymorphic aggregate is being defined.

`FINALFUNC_MODIFY` = { `READ_ONLY` | `SHAREABLE` | `READ_WRITE` }

This option specifies whether the final function is a pure function that does not modify its arguments. `READ_ONLY` indicates it does not; the other two values indicate that it may change the transition state value. See [Notes](#) below for more detail. The default is `READ_ONLY`, except for ordered-set aggregates, for which the default is `READ_WRITE`.

<combinefunc>

The <combinefunc> function may optionally be specified to allow the aggregate function to support partial aggregation. If provided, the <combinefunc> must combine two <state_data_type> values, each containing the result of aggregation over some subset of the input values, to produce a new <state_data_type> that represents the result of aggregating over both sets of inputs. This function can be thought of as an <sfunc>, where instead of acting upon an individual input row and adding it to the running aggregate state, it adds another aggregate state to the running state.

The <combinefunc> must be declared as taking two arguments of the <state_data_type> and returning a value of the <state_data_type>. Optionally this function may be “strict”. In this case the function will not be called when either of the input states are null; the other state will be taken as the correct result.

For aggregate functions whose <state_data_type> is `internal`, the <combinefunc> must not be strict. In this case the <combinefunc> must ensure that null states are handled correctly and that the state being returned is properly stored in the aggregate memory context.

<serialfunc>

An aggregate function whose <state_data_type> is `internal` can participate in parallel aggregation only if it has a <serialfunc> function, which must serialize the aggregate state into a `bytea` value for transmission to another process. This function must take a single argument of type `internal` and return type `bytea`. A corresponding <deserialfunc> is also required.

<deserialfunc>

Deserialize a previously serialized aggregate state back into <state_data_type>. This function must take two arguments of types `bytea` and `internal`, and produce a result of type `internal`. (Note: the second, `internal` argument is unused, but is required for type safety reasons.)

<initial_condition>

The initial setting for the state value. This must be a string constant in the form accepted for the data type <state_data_type>. If not specified, the state value starts out null.

<msfunc>

The name of the forward state transition function to be called for each input row in moving-aggregate mode. This is exactly like the regular transition function, except that its first argument and result are of type <mstate_data_type>, which might be different from <state_data_type>.

<minvfunc>

The name of the inverse state transition function to be used in moving-aggregate mode. This function has the same argument and result types as <msfunc>, but it is used to remove a value from the current aggregate state, rather than add a value to it. The inverse transition function must have the same strictness attribute as the forward state transition function.

<mstate_data_type>

The data type for the aggregate's state value, when using moving-aggregate mode.

<mstate_data_size>

The approximate average size (in bytes) of the aggregate's state value, when using moving-aggregate mode. This works the same as <state_data_size>, except that negative values are not used to indicate unbounded state size.

<mffunc>

The name of the final function called to compute the aggregate's result after all input rows have been traversed, when using moving-aggregate mode. This works the same as <ffunc>, except that its first argument's type is <mstate_data_type> and extra dummy arguments are specified by writing `MFINALFUNC_EXTRA`. The aggregate result type determined by <mffunc> or <mstate_data_type> must match that determined by the aggregate's regular implementation.

`MFINALFUNC_MODIFY` = { `READ_ONLY` | `SHAREABLE` | `READ_WRITE` }

This option is like `FINALFUNC_MODIFY`, but it describes the behavior of the moving-aggregate final function.

<initial_condition>

The initial setting for the state value, when using moving-aggregate mode. This works the same as <initial_condition>.

<sort_operator>

The associated sort operator for a `MIN` - or `MAX` -like aggregate. This is just an operator name (possibly schema-qualified). The operator is assumed to have the same input data types as the aggregate (which must be a single-argument normal aggregate).

<support_function>

The name (optionally schema-qualified) of a planner support function to use for this aggregate. See [???](#) for details. You must be superuser to use this option.

`PARALLEL` = { `SAFE` | `RESTRICTED` | `UNSAFE` }

The meanings of `PARALLEL SAFE`, `PARALLEL RESTRICTED`, and `PARALLEL UNSAFE` are the same as in [CREATE FUNCTION](#). An aggregate will not be considered for parallelization if it is marked `PARALLEL UNSAFE` (which is the default!) or `PARALLEL RESTRICTED`. Note that the parallel-safety markings of the aggregate's support functions are not consulted by the planner, only the marking of the aggregate itself.

`HYPOTHETICAL`

For ordered-set aggregates only, this flag specifies that the aggregate arguments are to be processed according to the requirements for hypothetical-set aggregates: that is, the last few direct arguments must match the data types of the aggregated (`WITHIN GROUP`) arguments. The `HYPOTHETICAL` flag has no effect on run-time behavior, only on parse-time resolution of the data types and collations of the aggregate's arguments.

The parameters of `CREATE AGGREGATE` can be written in any order, not just the order illustrated above.

Notes

In parameters that specify support function names, you can write a schema name if needed, for example `SFUNC = public.sum`. Do not write argument types there, however — the argument types of the support functions are determined from other parameters.

Ordinarily, PostgreSQL functions are expected to be true functions that do not modify their input values. However, an aggregate transition function, *when used in the context of an aggregate*, is allowed to cheat and modify its transition-state argument in place. This can provide substantial performance benefits compared to making a fresh copy of the transition state each time.

Likewise, while an aggregate final function is normally expected not to modify its input values, sometimes it is impractical to avoid modifying the transition-state argument. Such behavior must be declared using the `FINALFUNC_MODIFY` parameter. The `READ_WRITE` value indicates that the final function modifies the transition state in unspecified ways. This value prevents use of the aggregate as a window function, and it also prevents merging of transition states for aggregate calls that share the same input values and transition functions. The `SHAREABLE` value indicates that the transition function cannot be applied after the final function, but multiple final-function calls can be performed on the ending transition state value. This value prevents use of the aggregate as a window function, but it allows merging of transition states. (That is, the optimization of interest here is not applying the same final function repeatedly, but applying different final functions to the same ending transition state value. This is allowed as long as none of the final functions are marked `READ_WRITE`.)

If an aggregate supports moving-aggregate mode, it will improve calculation efficiency when the aggregate is used as a window function for a window with moving frame start (that is, a frame start mode other than `UNBOUNDED PRECEDING`). Conceptually, the forward transition function adds input values to the aggregate's state when they enter the window frame from the bottom, and the inverse transition function removes them again when they leave the frame at the top. So, when values are removed, they are always removed in the same order they were added. Whenever the inverse transition function is invoked, it will thus receive the earliest added but not yet removed argument value(s). The inverse transition function can assume that at least one row will remain in the current state after it removes the oldest row. (When this would not be the case, the window function mechanism simply starts a fresh aggregation, rather than using the inverse transition function.)

The forward transition function for moving-aggregate mode is not allowed to return NULL as the new state value. If the inverse transition function returns NULL, this is taken as an indication that the inverse function cannot reverse the state calculation for this particular input, and so the aggregate calculation will be redone from scratch for the current frame starting position. This convention allows moving-aggregate mode to be used in situations where there are some infrequent cases that are impractical to reverse out of the running state value.

If no moving-aggregate implementation is supplied, the aggregate can still be used with moving frames, but PostgreSQL will recompute the whole aggregation whenever the start of the frame moves. Note that whether or not the aggregate supports moving-aggregate mode, PostgreSQL can handle a moving frame end without recalculation; this is done by continuing to add new values to the aggregate's state. This is why use of an aggregate as a window function requires that the final function be read-only: it must not damage the aggregate's state value, so that the aggregation can be continued even after an aggregate result value has been obtained for one set of frame boundaries.

The syntax for ordered-set aggregates allows `VARIADIC` to be specified for both the last direct parameter and the last aggregated (`WITHIN GROUP`) parameter. However, the current implementation restricts use of `VARIADIC` in two ways. First, ordered-set aggregates can only use `VARIADIC "any"`, not other variadic array types. Second, if the last direct parameter is `VARIADIC "any"`, then there can be only one aggregated parameter and it must also be `VARIADIC "any"`. (In the representation used in the system catalogs, these two parameters are merged into a single `VARIADIC "any"` item, since `pg_proc` cannot represent functions with more than one `VARIADIC` parameter.) If the aggregate is a hypothetical-set aggregate, the direct arguments that match the `VARIADIC "any"` parameter are the hypothetical ones; any preceding parameters represent additional direct arguments that are not constrained to match the aggregated arguments.

Currently, ordered-set aggregates do not need to support moving-aggregate mode, since they cannot be used as window functions.

Partial (including parallel) aggregation is currently not supported for ordered-set aggregates. Also, it will never be used for aggregate calls that include `DISTINCT` or `ORDER BY` clauses, since those semantics cannot be supported during partial aggregation.

Examples

See [???](#).

Compatibility

`CREATE AGGREGATE` is a PostgreSQL language extension. The SQL standard does not provide for user-defined aggregate functions.

See Also

□□ #2

□ □□ □□ 2026-10-31 14:08:45 UTC

□ □□ □□ 2026-10-31 14:08:47 UTC