prepare



- Description
- Parameters
- Notes
- Examples
- Compatibility
- See Also

PREPARE

prepared statements

creating

PREPARE

7

SQL - Language Statements

PREPARE

prepare a statement for execution

PREPARE

name

[(

data_type

[, ...])] AS

statement

Description

`PREPARE` creates a prepared statement. A prepared statement is a server-side object that can be used to optimize performance. When the `PREPARE` statement is executed, the specified statement is parsed, analyzed, and rewritten. When an `EXECUTE` command is subsequently issued, the prepared statement is planned and executed. This division of labor avoids repetitive parse analysis work, while allowing the execution plan to depend on the specific parameter values supplied.

Prepared statements can take parameters: values that are substituted into the statement when it is executed. When creating the prepared statement, refer to parameters by position, using `$1`, `$2`, etc. A corresponding list of parameter data types can optionally be specified. When a parameter's data type is not specified or is declared as `unknown`, the type is inferred from the context in which the parameter is first referenced (if possible). When executing the statement, specify the actual values for these parameters in the `EXECUTE` statement. Refer to ??? for more information about that.

Prepared statements only last for the duration of the current database session. When the session ends, the prepared statement is forgotten, so it must be recreated before being used again. This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create their own prepared statement to use. Prepared statements can be manually cleaned up using the `DEALLOCATE` command.

Prepared statements potentially have the largest performance advantage when a single session is being used to execute a large number of similar statements. The performance difference will be particularly significant if the statements are complex to plan or rewrite, e.g., if the query involves a join of many tables or requires the application of several rules. If the statement is relatively simple to plan and rewrite but relatively expensive to execute, the performance advantage of prepared statements will be less noticeable.

Parameters

<name>
An arbitrary name given to this particular prepared statement. It must be unique within a single session and is subsequently used to execute or deallocate a previously prepared statement.

<data_type>
The data type of a parameter to the prepared statement. If the data type of a particular parameter is unspecified or is specified as `unknown`, it will be inferred from the context in which the parameter is first referenced. To refer to the parameters in the prepared statement itself, use `$1`, `$2`, etc.

<statement>
Any `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`, or `VALUES` statement.

Notes

A prepared statement can be executed with either a generic plan or a custom plan. A generic plan is the same across all executions, while a custom plan is generated for a specific execution using the parameter values given in that call. Use of a generic plan avoids planning overhead, but in some situations a custom plan will be much more efficient to execute because the planner can make use of knowledge of the parameter values. (Of course, if the prepared statement has no parameters, then this is moot and a generic plan is always used.)

By default (that is, when ??? is set to `auto`), the server will automatically choose whether to use a generic or custom plan for a prepared statement that has parameters. The current rule for this is that the first five executions are done with custom plans and the average estimated cost of those plans is calculated. Then a generic plan is created and its estimated cost is compared to the average custom-plan cost. Subsequent executions use the generic plan if its cost is not so much higher than the average custom-plan cost as to make repeated replanning seem preferable.

This heuristic can be overridden, forcing the server to use either generic or custom plans, by setting `plan_cache_mode` to `force_generic_plan` or `force_custom_plan` respectively. This setting is primarily useful if the generic plan's cost estimate is badly off for some reason, allowing it to be chosen even though its actual cost is much more than that of a custom plan.

To examine the query plan PostgreSQL is using for a prepared statement, use `EXPLAIN`, for example

```
EXPLAIN EXECUTE name(parameter_values);
```

If a generic plan is in use, it will contain parameter symbols `$n`, while a custom plan will have the supplied parameter values substituted into it.

For more information on query planning and the statistics collected by PostgreSQL for that purpose, see the ??? documentation.

Although the main point of a prepared statement is to avoid repeated parse analysis and planning of the statement, PostgreSQL will force re-analysis and re-planning of the statement before using it whenever database objects used in the statement have undergone definitional (DDL) changes or their planner statistics have been updated since the previous use of the prepared statement. Also, if the value of ??? changes from one use to the next, the statement will be re-parsed using the new `search_path`. (This latter behavior is new as of PostgreSQL 9.3.) These rules make use of a prepared statement semantically almost equivalent to re-submitting the same query text over and over, but with a performance benefit if no object definitions are changed, especially if the best plan

remains the same across uses. An example of a case where the semantic equivalence is not perfect is that if the statement refers to a table by an unqualified name, and then a new table of the same name is created in a schema appearing earlier in the `search_path`, no automatic re-parse will occur since no object used in the statement changed. However, if some other change forces a re-parse, the new table will be referenced in subsequent uses.

You can see all prepared statements available in the session by querying the pg_prepared_statements system view.

Examples

Create a prepared statement for an `INSERT` statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Create a prepared statement for a `SELECT` statement, and then execute it:

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

In this example, the data type of the second parameter is not specified, so it is inferred from the context in which `$2` is used.

Compatibility

The SQL standard includes a `PREPARE` statement, but it is only for use in embedded SQL. This version of the `PREPARE` statement also uses a somewhat different syntax.

See Also

□□□ #2
□ □□□ □□□ 2026-10-31 14:13:51 UTC
□ □□□ □□□ 2026-10-31 14:13:52 UTC