create_statistics

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

CREATE STATISTICS

CREATE STATISTICS

7

SQL - Language Statements

CREATE STATISTICS

define extended statistics

CREATE STATISTICS [[IF NOT EXISTS]

statistics_name

] ON {

column_name

| (

expression

) } FROM

table_name

CREATE STATISTICS [[IF NOT EXISTS]

statistics_name

] [(

statistics_kind

[, ...])] ON {

column_name

| (

expression

) }, {

column_name

| (

expression

) } [, ...] FROM

table_name

Description

`CREATE STATISTICS` will create a new extended statistics object tracking data about the specified table, foreign table or materialized view. The statistics object will be created in the current database and will be owned by the user issuing the command.

The `CREATE STATISTICS` command has two basic forms. The first form allows univariate statistics for a single expression or virtual generated column to be collected, providing benefits similar to an expression index without the overhead of index maintenance. This form does not allow the statistics kind to be specified, since the various statistics kinds refer only to multivariate statistics. The second form of the command allows multivariate statistics on multiple columns and/or expressions to be collected, optionally specifying which statistics kinds to include. This form will also automatically cause univariate statistics to be collected on any expressions and virtual generated columns included in the list.

If a schema name is given (for example, `CREATE STATISTICS myschema.mystat ...`) then the statistics object is created in the specified schema. Otherwise it is created in the current schema. If given, the name of the statistics object must be distinct from the name of any other statistics object in the

same schema.

Parameters

`IF NOT EXISTS`
Do not throw an error if a statistics object with the same name already exists. A notice is issued in this case. Note that only the name of the statistics object is considered here, not the details of its definition. Statistics name is required when `IF NOT EXISTS` is specified.

<statistics_name>
The name (optionally schema-qualified) of the statistics object to be created. If the name is omitted, PostgreSQL chooses a suitable name based on the parent table's name and the defined column name(s) and/or expression(s).

<statistics_kind>
A multivariate statistics kind to be computed in this statistics object. Currently supported kinds are `ndistinct`, which enables n-distinct statistics, `dependencies`, which enables functional dependency statistics, and `mcv` which enables most-common values lists. If this clause is omitted, all supported statistics kinds are included in the statistics object. Univariate expression statistics are built automatically if the statistics definition includes any complex expressions or references to virtual generated columns rather than just simple column references. For more information, see ??? and ???.

<column_name>
The name of a table column to be covered by the computed statistics. This may be used to build univariate statistics on a single virtual generated column, or as part of a list of multiple columns (virtual or non-virtual) and/or expressions to build multivariate statistics. In the latter case, separate univariate statistics are built automatically for each expression and virtual generated column in the list.

Defining extended statistics on a single *non-virtual* column is not supported or necessary, because statistics are built automatically on such columns without defining extended statistics.

<expression>
An expression to be covered by the computed statistics. This may be used to build univariate statistics on a single expression, or as part of a list of multiple column names and/or expressions to build multivariate statistics. In the latter case, separate univariate statistics are built automatically for each expression and virtual generated column in the list.

<table_name>
The name (optionally schema-qualified) of the table containing the column(s) the statistics are computed on; see ??? for an explanation of the handling of inheritance and partitions.

Notes

You must be the owner of a table to create a statistics object reading it. Once created, however, the ownership of the statistics object is independent of the underlying table(s).

Expression statistics are per-expression and are similar to creating an index on the expression, except that they avoid the overhead of index maintenance. Expression statistics are built automatically for each expression in the statistics object definition. Extended statistics on a virtual generated column behave the same as expression statistics on the column's generation expression.

Extended statistics are not currently used by the planner for selectivity estimations made for table joins. This limitation will likely be removed in a future version of PostgreSQL.

Examples

Create table t1 with two functionally dependent columns, i.e., knowledge of a value in the first column is sufficient for determining the value in the other column. Then functional dependency statistics are built on those columns:

```
CREATE TABLE t1 (
    a   int,
    b   int
);

INSERT INTO t1 SELECT i/100, i/500
                FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- the number of matching rows will be drastically underestimated:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;

ANALYZE t1;

-- now the row count estimate is more accurate:
```

```
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

Without functional-dependency statistics, the planner would assume that the two `WHERE` conditions are independent, and would multiply their selectivities together to arrive at a much-too-small row count estimate. With such statistics, the planner recognizes that the `WHERE` conditions are redundant and does not underestimate the row count.

Create table t2 with two perfectly correlated columns (containing identical data), and an MCV list on those columns:

```
CREATE TABLE t2 (
    a   int,
    b   int
);

INSERT INTO t2 SELECT mod(i,100), mod(i,100)
                FROM generate_series(1,1000000) s(i);

CREATE STATISTICS s2 (mcv) ON a, b FROM t2;

ANALYZE t2;

-- valid combination (found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 1);

-- invalid combination (not found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 2);
```

The MCV list gives the planner more detailed information about the specific values that commonly appear in the table, as well as an upper bound on the selectivities of combinations of values that do not appear in the table, allowing it to generate better estimates in both cases.

Create table t3 with a single timestamp column, and run queries using expressions on that column. Without extended statistics, the planner has no information about the data distribution for the expressions, and uses default estimates. The planner also does not realize that the value of the date truncated to the month is fully determined by the value of the date truncated to the day. Then expression and ndistinct statistics are built on those two expressions:

```
CREATE TABLE t3 (
    a   timestamp
);
```

```
INSERT INTO t3 SELECT i FROM generate_series('2020-01-01'::timestamp,
                                            '2020-12-31'::timestamp,
                                            '1 minute'::interval) s(i);

ANALYZE t3;

-- the number of matching rows will be drastically underestimated:
EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('month', a) = '2020-01-01'::timestamp;

EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('day', a) BETWEEN '2020-01-01'::timestamp
                                 AND '2020-06-30'::timestamp;

EXPLAIN ANALYZE SELECT date_trunc('month', a), date_trunc('day', a)
   FROM t3 GROUP BY 1, 2;

-- build ndistinct statistics on the pair of expressions (per-expression
-- statistics are built automatically)
CREATE STATISTICS s3 (ndistinct) ON date_trunc('month', a), date_trunc('day', a) FROM t3;

ANALYZE t3;

-- now the row count estimates are more accurate:
EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('month', a) = '2020-01-01'::timestamp;

EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('day', a) BETWEEN '2020-01-01'::timestamp
                                 AND '2020-06-30'::timestamp;

EXPLAIN ANALYZE SELECT date_trunc('month', a), date_trunc('day', a)
   FROM t3 GROUP BY 1, 2;
```

Without expression and ndistinct statistics, the planner has no information about the number of distinct values for the expressions, and has to rely on default estimates. The equality and range conditions are assumed to have 0.5% selectivity, and the number of distinct values in the expression is assumed to be the same as for the column (i.e. unique). This results in a significant underestimate of the row count in the first two queries. Moreover, the planner has no information about the relationship between the expressions, so it assumes the two `WHERE` and `GROUP BY` conditions are independent, and multiplies their selectivities together to arrive at a severe

overestimate of the group count in the aggregate query. This is further exacerbated by the lack of accurate statistics for the expressions, forcing the planner to use a default ndistinct estimate for the expression derived from ndistinct for the column. With such statistics, the planner recognizes that the conditions are correlated, and arrives at much more accurate estimates.

Compatibility

There is no `CREATE STATISTICS` command in the SQL standard.

See Also

□□□ #2
□ □□□ □□□ 2026-10-31 14:09:49 UTC
□ □□□ □□□ 2026-10-31 14:09:49 UTC