pgTAP 0.96.0

pgTAP is a unit testing framework for PostgreSQL written in PL/pgSQL and PL/SQL. It includes a comprehensive collection of TAP-emitting assertion functions, as well as the ability to integrate with other TAP-emitting test frameworks. It can also be used in the xUnit testing style.

Contents

home
download
documentation
pg_prove
integration
mail lists
github

code: david e. wheeler
webdesign: tri-star
photo: courtland whited

Synopsis

```
SELECT plan( 23 );
-- or SELECT * from no_plan();

-- Various ways to say "ok"
SELECT ok( :have = :want, :test_description );

SELECT is(   :have, :want, :test_description );
SELECT isnt( :have, :want, :test_description );

-- Rather than \echo # here's what went wrong
SELECT diag( 'here''s what went wrong' );

-- Compare values with LIKE or regular expressions.
SELECT alike(   :have, :like_expression, :test_description );
SELECT unalike( :have, :like_expression, :test_description );

SELECT matches(      :have, :regex, :test_description );
SELECT doesnt_match( :have, :regex, :test_description );

SELECT cmp_ok(:have, '=', :want, :test_description );

-- Skip tests based on runtime conditions.
SELECT CASE WHEN :some_feature THEN collect_tap(
    ok( foo(),       :test_description),
    is( foo(42), 23, :test_description)
) ELSE skip(:why, :how_many ) END;

-- Mark some tests as to-do tests.
SELECT todo(:why, :how_many);
SELECT ok( foo(),       :test_description);
SELECT is( foo(42), 23, :test_description);

-- Simple pass/fail.
SELECT pass(:test_description);
SELECT fail(:test_description);
```

Installation

For the impatient, to install pgTAP into a PostgreSQL database, just do this:

```
make
make install
make installcheck
```

If you encounter an error such as:

```
"Makefile", line 8: Need an operator
```

You need to use GNU make, which may well be installed on your system as 'gmake':

```
gmake
gmake install
gmake installcheck
```

If you encounter an error such as:

```
make: pg_config: Command not found
```

Or:

```
Makefile:52: *** pgTAP requires PostgreSQL 8.1 or later. This is .  S
```

Be sure that you have `pg_config` installed and in your path. If you used a package management system such as RPM to install PostgreSQL, be sure that the `-devel` package is also installed. If necessary tell the build process where to find it:

```
env PG_CONFIG=/path/to/pg_config make && make install && make install
```

And finally, if all that fails (and if you're on PostgreSQL 8.1, it likely will), copy the entire distribution directory to the `contrib/` subdirectory of the PostgreSQL source tree and try it there without `pg_config`:

```
env NO_PGXS=1 make && make install && make installcheck
```

If you encounter an error such as:

```
ERROR:  must be owner of database regression
```

You need to run the test suite using a super user, such as the default "postgres" super user:

```
make installcheck PGUSER=postgres
```

Once pgTAP is installed, you can add it to a database. If you're running PostgreSQL 9.1.0 or greater, it's a simple as connecting to a database as a super user and

running:

```
CREATE EXTENSION pgtap;
```

If you've upgraded your cluster to PostgreSQL 9.1 and already had pgTAP installed, you can upgrade it to a properly packaged extension with:

```
CREATE EXTENSION pgtap FROM unpackaged;
```

For versions of PostgreSQL less than 9.1.0, you'll need to run the installation script:

```
psql -d mydb -f /path/to/pgsql/share/contrib/pgtap.sql
```

If you want to install pgTAP and all of its supporting objects into a specific schema, use the `PGOPTIONS` environment variable to specify the schema, like so:

```
PGOPTIONS=--search_path=tap psql -d mydb -f pgTAP.sql
```

Testing pgTAP with pgTAP

In addition to the PostgreSQL-standard `installcheck` target, the `test` target uses the `pg_prove` Perl program to do its testing, which will be installed with the TAP::Parser::SourceHandler::pgTAP CPAN distribution. You'll need to make sure that you use a database with PL/pgSQL loaded, or else the tests won't work. `pg_prove` supports a number of environment variables that you might need to use, including all the usual PostgreSQL client environment variables:

- `$PGDATABASE`
- `$PGHOST`
- `$PGPORT`
- `$PGUSER`

You can use it to run the test suite as a database super user like so:

```
make test PGUSER=postgres
```

Adding pgTAP to a Database

Once pgTAP has been built and tested, you can install it into a PL/pgSQL-enabled database:


```
psql -d dbname -f pgtap.sql
```

If you want pgTAP to be available to all new databases, install it into the "template1" database:

```
psql -d template1 -f pgtap.sql
```

If you want to remove pgTAP from a database, run the `uninstall_pgtap.sql` script:

```
psql -d dbname -f uninstall_pgtap.sql
```

Both scripts will also be installed in the `contrib` directory under the directory output by `pg_config --sharedir`. So you can always do this:

```
psql -d template1 -f `pg_config --sharedir`/contrib/pgtap.sql
```

But do be aware that, if you've specified a schema using `$TAPSCHEMA`, that schema will always be created and the pgTAP functions placed in it.

pgTAP Test Scripts

You can distribute `pgtap.sql` with any PostgreSQL distribution, such as a custom data type. For such a case, if your users want to run your test suite using PostgreSQL's standard `installcheck` make target, just be sure to set variables to keep the tests quiet, start a transaction, load the functions in your test script, and then rollback the transaction at the end of the script. Here's an example:

```
\set ECHO
\set QUIET 1
-- Turn off echo and keep things quiet.

-- Format the output for nice TAP.
\pset format unaligned
\pset tuples_only true
\pset pager

-- Revert all changes on failure.
\set ON_ERROR_ROLLBACK 1
\set ON_ERROR_STOP true
\set QUIET 1

-- Load the TAP functions.
BEGIN;
```


```
    \i pgtap.sql

-- Plan the tests.
SELECT plan(1);

-- Run the tests.
SELECT pass( 'My test passed, w00t!' );

-- Finish the tests and clean up.
SELECT * FROM finish();
ROLLBACK;
```

Of course, if you already have the pgTAP functions in your testing database, you should skip `\i pgtap.sql` at the beginning of the script.

The only other limitation is that the `pg_typeof()` function, which is written in C, will not be available in 8.3 and lower. You'll want to comment out its declaration in the bundled copy of `pgtap.sql` and then avoid using `cmp_ok()`, since that function relies on `pg_typeof()`. Note that `pg_typeof()` is included in PostgreSQL 8.4, so you won't need to avoid it on that version or higher.

Now you're ready to run your test script!

```
% psql -d try -Xf test.sql
1..1
ok 1 - My test passed, w00t!
```

You'll need to have all of those variables in the script to ensure that the output is proper TAP and that all changes are rolled back – including the loading of the test functions – in the event of an uncaught exception.

Using `pg_prove`

Or save yourself some effort – and run a batch of tests scripts or all of your xUnit test functions at once – by using `pg_prove`, available in the TAP::Parser::SourceHandler::pgTAP CPAN distribution. If you're not relying on `installcheck`, your test scripts can be a lot less verbose; you don't need to set all the extra variables, because `pg_prove` takes care of that for you:

```
-- Start transaction and plan the tests.
BEGIN;
SELECT plan(1);

-- Run the tests.
```


```
SELECT pass( 'My test passed, w00t!' );

-- Finish the tests and clean up.
SELECT * FROM finish();
ROLLBACK;
```

Now run the tests. Here's what it looks like when the pgTAP tests are run with `pg_prove`:

```
% pg_prove -U postgres sql/*.sql
sql/coltap.....ok
sql/hastap.....ok
sql/moretap....ok
sql/pg73.......ok
sql/pktap......ok
All tests successful.
Files=5, Tests=216,  1 wallclock secs ( 0.06 usr  0.02 sys +  0.08 cu
Result: PASS
```

If you're using xUnit tests and just want to have `pg_prove` run them all through the `runtests()` function, just tell it to do so:

```
% pg_prove -d myapp --runtests
```

Yep, that's all there is to it. Call `pg_prove --verbose` to see the individual test descriptions, `pg_prove --help` to see other supported options, and `pg_prove --man` to see its entire documentation.

Using pgTAP

The purpose of pgTAP is to provide a wide range of testing utilities that output TAP. TAP, or the "Test Anything Protocol", is an emerging standard for representing the output from unit tests. It owes its success to its format as a simple text-based interface that allows for practical machine parsing and high legibility for humans. TAP started life as part of the test harness for Perl but now has implementations in C/C++, Python, PHP, JavaScript, Perl, and now PostgreSQL.

There are two ways to use pgTAP: 1) In simple test scripts that use a plan to describe the tests in the script; or 2) In xUnit-style test functions that you install into your database and run all at once in the PostgreSQL client of your choice.

I love it when a plan comes together

Before anything else, you need a testing plan. This basically declares how many tests your script is going to run to protect against premature failure.

The preferred way to do this is to declare a plan by calling the `plan()` function:

```
SELECT plan( 42 );
```

There are rare cases when you will not know beforehand how many tests your script is going to run. In this case, you can declare that you have no plan. (Try to avoid using this as it weakens your test.)

```
SELECT * FROM no_plan();
```

Often, though, you'll be able to calculate the number of tests, like so:

```
SELECT plan( COUNT(*) )
  FROM foo;
```

At the end of your script, you should always tell pgTAP that the tests have completed, so that it can output any diagnostics about failures or a discrepancy between the planned number of tests and the number actually run:

```
SELECT * FROM finish();
```

What a sweet unit!

If you're used to xUnit testing frameworks, you can collect all of your tests into database functions and run them all at once with `runtests()`. This is similar to how PGUnit and Epic work. The `runtests()` function does all the work of finding and running your test functions in individual transactions. It even supports setup and teardown functions. To use it, write your unit test functions so that they return a set of text results, and then use the pgTAP assertion functions to return TAP values, like so:

```
CREATE OR REPLACE FUNCTION setup_insert(
) RETURNS SETOF TEXT AS $$
    RETURN NEXT is( MAX(nick), NULL, 'Should have no users') FROM use
    INSERT INTO users (nick) VALUES ('theory');
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION test_user(
) RETURNS SETOF TEXT AS $$
   SELECT is( nick, 'theory', 'Should have nick') FROM users;
```


```
END;
$$ LANGUAGE sql;
```

See below for details on the pgTAP assertion functions. Once you've defined your unit testing functions, you can run your tests at any time using the `runtests()` function:

```
SELECT * FROM runtests();
```

Each test function will run within its own transaction, and rolled back when the function completes (or after any teardown functions have run). The TAP results will be sent to your client.

Test names

By convention, each test is assigned a number in order. This is largely done automatically for you. However, it's often very useful to assign a name to each test. Would you rather see this?

```
  ok 4
  not ok 5
  ok 6
```

Or this?

```
  ok 4 - basic multi-variable
  not ok 5 - simple exponential
  ok 6 - force == mass * acceleration
```

The latter gives you some idea of what failed. It also makes it easier to find the test in your script, simply search for "simple exponential".

All test functions take a name argument. It's optional, but highly suggested that you use it.

Sometimes it's useful to extract test function names from pgtap output, especially when using xUnit style with Continuous Integration Server like Hudson or TeamCity. By default pgTAP displays this names as "comment", but you're able to change this behavior by overriding function `diag_test_name`:

diag_test_name()


```
CREATE OR REPLACE FUNCTION diag_test_name(TEXT)
RETURNS TEXT AS $$
    SELECT diag('test: ' || $1 );
$$ LANGUAGE SQL;
```

Parameters

`:test_name`

A test name.

This will show # test: my_example_test_function_name instead of # my_example_test_function_name() This makes easy handling test name and differing test names from comments.

I'm ok, you're not ok

The basic purpose of pgTAP–and of any TAP-emitting test framework, for that matter–is to print out either "ok #" or "not ok #", depending on whether a given test succeeded or failed. Everything else is just gravy.

All of the following functions return "ok" or "not ok" depending on whether the test succeeded or failed.

ok()

```
SELECT ok( :boolean, :description );
SELECT ok( :boolean );
```

Parameters

`:boolean`

A boolean value indicating success or failure.

`:description`

A short description of the test.

This function simply evaluates any boolean expression and uses it to determine if the test succeeded or failed. A true expression passes, a false one fails. Very simple.

For example:

```
SELECT ok( 9 ^ 2 = 81,    'simple exponential' );
SELECT ok( 9 < 10,        'simple comparison' );
SELECT ok( 'foo' ~ '^f',  'simple regex' );
```


```
SELECT ok( active = true, name ||  widget active' )
  FROM widgets;
```

(Mnemonic: “This is ok.”)

The `:description` is a very short description of the test that will be printed out. It makes it very easy to find a test in your script when it fails and gives others an idea of your intentions. The description is optional, but we *very* strongly encourage its use.

Should an `ok()` fail, it will produce some diagnostics:

```
not ok 18 - sufficient mucus
#     Failed test 18: "sufficient mucus"
```

Furthermore, should the boolean test result argument be passed as a `NULL` rather than `true` or `false`, `ok()` will assume a test failure and attach an additional diagnostic:

```
not ok 18 - sufficient mucus
#     Failed test 18: "sufficient mucus"
#     (test result was NULL)
```

is()
isnt()

```
SELECT is(   :have, :want, :description );
SELECT is(   :have, :want );
SELECT isnt( :have, :want, :description );
SELECT isnt( :have, :want );
```

Parameters

`:have`

Value to test.

`:want`

Value that `:have` is expcted to be. Must be the same data type.

`:description`

A short description of the test.

Similar to `ok()`, `is()` and `isnt()` compare their two arguments with `IS NOT DISTINCT FROM` (=) AND `IS DISTINCT FROM` (<>) respectively and use the result of that to determine if the test succeeded or failed. So these:


```
-- Is the ultimate answer 42?
SELECT is( ultimate_answer(), 42, 'Meaning of Life' );

-- foo() doesn't return empty
SELECT isnt( foo(), '', 'Got some foo' );
```

are similar to these:

```
SELECT ok(   ultimate_answer() =  42, 'Meaning of Life' );
SELECT isnt( foo()             <> '', 'Got some foo'    );
```

(Mnemonic: “This is that.” “This isn’t that.”)

Note: Thanks to the use of the `IS [ NOT ] DISTINCT FROM` construct, `NULL`s are not treated as unknowns by `is()` or `isnt()`. That is, if `:have` and `:want` are both `NULL`, the test will pass, and if only one of them is `NULL`, the test will fail.

So why use these test functions? They produce better diagnostics on failure. `ok()` cannot know what you are testing for (beyond the description), but `is()` and `isnt()` know what the test was and why it failed. For example this test:

```
\set foo '\'waffle\''
\set bar '\'yarblokos\''
SELECT is( :foo::text, :bar::text, 'Is foo the same as bar?' );
```

Will produce something like this:

```
# Failed test 17:  "Is foo the same as bar?"
#         have: waffle
#         want: yarblokos
```

So you can figure out what went wrong without re-running the test.

You are encouraged to use `is()` and `isnt()` over `ok()` where possible. You can even use them to compar records in PostgreSQL 8.4 and later:

```
SELECT is( users.*, ROW(1, 'theory', true)::users )
  FROM users
 WHERE nick = 'theory';
```

matches()

```
SELECT matches( :have, :regex, :description );
SELECT matches( :have, :regex );
```

Parameters

`:have`

Value to match.

`:regex`

A regular expression.

`:description`

A short description of the test.

Similar to `ok()`, `matches()` matches `:have` against the regex `:regex`.

So this:

```
SELECT matches( :this, '^that', 'this is like that' );
```

is similar to:

```
SELECT ok( :this ~ '^that', 'this is like that' );
```

(Mnemonic “This matches that”.)

Its advantages over `ok()` are similar to that of `is()` and `isnt()`: Better diagnostics on failure.

`imatches()`

```
SELECT imatches( :have, :regex, :description );
SELECT imatches( :have, :regex );
```

Parameters

`:have`

Value to match.

`:regex`

A regular expression.

`:description`

A short description of the test.

Just like `matches()` except that the regular expression is compared to `:have` case-insensitively.

`doesnt_match()`

`doesnt_imatch()`

```
SELECT doesnt_match(  :have, :regex, :description );
SELECT doesnt_match(  :have, :regex );
```


```
SELECT doesnt_imatch( :have, :regex, :description );
SELECT doesnt_imatch( :have, :regex );
```

Parameters

`:have`

Value to match.

`:regex`

A regular expression.

`:description`

A short description of the test.

These functions work exactly as `matches()` and `imatches()` do, only they check if `:have` *does not* match the given pattern.

alike()
ialike()

```
SELECT alike(  :this, :like, :description );
SELECT alike(  :this, :like );
SELECT ialike( :this, :like, :description );
SELECT ialike( :this, :like );
```

Parameters

`:have`

Value to match.

`:like`

A SQL `LIKE` pattern.

`:description`

A short description of the test.

Similar to `matches()`, `alike()` matches `:hve` against the SQL `LIKE` pattern `:like`. `ialike()` matches case-insensitively.

So this:

```
SELECT ialike( :have, 'that%', 'this is alike that' );
```

is similar to:

```
SELECT ok( :have ILIKE 'that%', 'this is like that' );
```

(Mnemonic “This is like that”.)

Its advantages over `ok()` are similar to that of `is()` and `isnt()`: Better diagnostics on failure.

unalike()
unialike()

```
SELECT unalike(  :this, :like, :description );
SELECT unalike(  :this, :like );
SELECT unialike( :this, :like, :description );
SELECT unialike( :this, :like );
```

Parameters

`:have`

Value to match.

`:like`

A SQL `LIKE` pattern.

`:description`

A short description of the test.

Works exactly as `alike()`, only it checks if `:have` *does not* match the given pattern.

cmp_ok()

```
SELECT cmp_ok( :have, :op, :want, :description );
SELECT cmp_ok( :have, :op, :want );
```

Parameters

`:have`

Value to compare.

`:op`

An SQL operator specified as a string.

`:want`

Value to compare to `:have` using the `:op` operator.

`:description`

A short description of the test.

Halfway between `ok()` and `is()` lies `cmp_ok()`. This function allows you to compare two arguments using any binary operator.


```
-- ok( :have = :want );
SELECT cmp_ok( :have, '=', :want, 'this = that' );

-- ok( :have >= :want );
SELECT cmp_ok( :have, '>=', :want, 'this >= that' );

-- ok( :have && :want );
SELECT cmp_ok( :have, '&&', :want, 'this && that' );
```

Its advantage over `ok()` is that when the test fails you'll know what `:have` and `:want` were:

```
not ok 1
#     Failed test 1:
#     '23'
#         &&
#     NULL
```

Note that if the value returned by the operation is `NULL`, the test will be considered to have failed. This may not be what you expect if your test was, for example:

```
SELECT cmp_ok( NULL, '=', NULL );
```

But in that case, you should probably use `is()`, instead.

pass()
fail()

```
SELECT pass( :description );
SELECT pass( );
SELECT fail( :description );
SELECT fail( );
```

Parameters

`:description`

A short description of the test.

Sometimes you just want to say that the tests have passed. Usually the case is you've got some complicated condition that is difficult to wedge into an `ok()`. In this case, you can simply use `pass()` (to declare the test ok) or `fail()` (for not ok). They are synonyms for `ok(1)` and `ok(0)`.

Use these functions very, very, very sparingly.

isa_ok()

```
SELECT isa_ok( :have, :regtype, :name );
SELECT isa_ok( :have, :regtype );
```

Parameters

`:have`

Value to check the type of.

`:regtype`

Name of an SQL data type.

`:name`

A name for the value being compared.

Checks to see if the given value is of a particular type. The description and diagnostics of this test normally just refer to “the value”. If you’d like them to be more specific, you can supply a `:name`. For example you might say “the return value” when yo’re examing the result of a function call:

```
SELECT isa_ok( length('foo'), 'integer', 'The return value from lengt
```

In which case the description will be “The return value from length() isa integer”.

In the event of a failure, the diagnostic message will tell you what the type of the value actually is:

```
not ok 12 - the value isa integer[]
#     the value isn't a "integer[]" it's a "boolean"
```

Pursuing Your Query

Sometimes, you’ve just gotta test a query. I mean the results of a full blown query, not just the scalar assertion functions we’ve seen so far. pgTAP provides a number of functions to help you test your queries, each of which takes one or two SQL statements as arguments. For example:

```
SELECT throws_ok('SELECT divide_by(0)');
```

Yes, as strings. Of course, you’ll often need to do something complex in your SQL, and quoting SQL in strings in what is, after all, an SQL application, is an unnecessary PITA. Each of the query-executing functions in this section thus

support an alternative to make your tests more SQLish: using prepared statements.

Prepared statements allow you to just write SQL and simply pass the prepared statement names to test functions. For example, the above example can be rewritten as:

```
PREPARE mythrow AS SELECT divide_by(0);
SELECT throws_ok('mythrow');
```

pgTAP assumes that an SQL argument without space characters or starting with a double quote character is a prepared statement and simply `EXECUTE`s it. If you need to pass arguments to a prepared statement, perhaps because you plan to use it in multiple tests to return different values, just `EXECUTE` it yourself. Here's an example with a prepared statement with a space in its name, and one where arguments need to be passed:

```
PREPARE "my test" AS SELECT * FROM active_users() WHERE name LIKE 'A%
PREPARE expect AS SELECT * FROM users WHERE active = $1 AND name LIKE

SELECT results_eq(
    '"my test"',
    'EXECUTE expect( true, ''A%'' )'
);
```

Since "my test" was declared with double quotes, it must be passed with double quotes. And since the call to "expect" included spaces (to keep it legible), the `EXECUTE` keyword was required.

In PostgreSQL 8.2 and up, you can also use a `VALUES` statement, both in the query string or in a prepared statement. A useless example:

```
PREPARE myvals AS VALUES (1, 2), (3, 4);
SELECT set_eq(
    'myvals',
    'VALUES (1, 2), (3, 4)'
);
```

Here's a bonus if you need to check the results from a query that returns a single column: for those functions that take two query arguments, the second can be an array. Check it out:


```
SELECT results_eq(
    'SELECT * FROM active_user_ids()',
    ARRAY[ 2, 3, 4, 5]
);
```

The first query *must* return only one column of the same type as the values in the array. If you need to test more columns, you'll need to use two queries.

Keeping these techniques in mind, read on for all of the query-testing goodness.

To Error is Human

Sometimes you just want to know that a particular query will trigger an error. Or maybe you want to make sure a query *does not* trigger an error. For such cases, we provide a couple of test functions to make sure your queries are as error-prone as you think they should be.

`throws_ok()`

```
SELECT throws_ok( :sql, :errcode, :ermsg, :description );
SELECT throws_ok( :sql, :errcode, :ermsg );
SELECT throws_ok( :sql, :errcode );
SELECT throws_ok( :sql, :errmsg, :description );
SELECT throws_ok( :sql, :errmsg );
SELECT throws_ok( :sql );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:errcode`

A PostgreSQL error code

`:errmsg`

An error message.

`:description`

A short description of the test.

When you want to make sure that an exception is thrown by PostgreSQL, use `throws_ok()` to test for it.

The first argument should be the name of a prepared statement or else a string representing the query to be executed (see the summary for query argument

details). `throws_ok()` will use the PL/pgSQL `EXECUTE` statement to execute the query and catch any exception.

The second argument should be an exception error code, which is a five-character string (if it happens to consist only of numbers and you pass it as an integer, it will still work). If this value is not `NULL`, `throws_ok()` will check the thrown exception to ensure that it is the expected exception. For a complete list of error codes, see Appendix A. in the PostgreSQL documentation.

The third argument is an error message. This will be most useful for functions you've written that raise exceptions, so that you can test the exception message that you've thrown. Otherwise, for core errors, you'll need to be careful of localized error messages. One trick to get around localized error messages is to pass NULL as the third argument. This allows you to still pass a description as the fourth argument.

The fourth argument is of course a brief test description. Here's a useful example:

```
PREPARE my_thrower AS INSERT INTO try (id) VALUES (1);
SELECT throws_ok(
    'my_thrower',
    '23505',
    'duplicate key value violates unique constraint "try_pkey"',
    'We should get a unique violation for a duplicate PK'
);
```

For the two- and three-argument forms of `throws_ok()`, if the second argument is exactly five bytes long, it is assumed to be an error code and the optional third argument is the error message. Otherwise, the second argument is assumed to be an error message and the third argument is a description. If for some reason you need to test an error message that is five bytes long, use the four-argument form.

A failing `throws_ok()` test produces an appropriate diagnostic message. For example:

```
# Failed test 81: "This should die a glorious death"
#       caught: 23505: duplicate key value violates unique constraint
#       wanted: 23502: null value in column "id" violates not-null co
```

Idea borrowed from the Test::Exception Perl module.

throws_like()

throws_ilike()

```
SELECT throws_like(  :sql, :like, :description );
SELECT throws_like(  :sql, :like );
SELECT throws_ilike( :sql, :like, :description );
SELECT throws_ilike( :sql, :like );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:like`

An SQL `LIKE` pattern.

`:description`

A short description of the test.

Like `throws_ok()`, but tests that an exception error message matches an SQL `LIKE` pattern. The `throws_ilike()` variant matches case-insensitively. An example:

```
PREPARE my_thrower AS INSERT INTO try (tz) VALUES ('America/Moscow');
SELECT throws_like(
    'my_thrower',
    '%"timezone_check"',
    'We should error for invalid time zone'
);
```

A failing `throws_like()` test produces an appropriate diagnostic message. For example:

```
# Failed test 85: "We should error for invalid time zone"
#     error message: 'value for domain timezone violates check constr
#     doesn't match: '%"timezone_check"'
```

throws_matching()
throws_imatching()

```
SELECT throws_matching(  :sql, :regex, :description );
SELECT throws_matching(  :sql, :regex );
SELECT throws_imatching( :sql, :regex, :description );
SELECT throws_imatching( :sql, :regex );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:regex`

A regular expression.

`:description`

A short description of the test.

Like `throws_ok()`, but tests that an exception error message matches a regular expression. The `throws_imatching()` variant matches case-insensitively. An example:

```
PREPARE my_thrower AS INSERT INTO try (tz) VALUES ('America/Moscow');
SELECT throws_matching(
    'my_thrower',
    '.+"timezone_check"',
    'We should error for invalid time zone'
);
```

A failing `throws_matching()` test produces an appropriate diagnostic message. For example:

```
# Failed test 85: "We should error for invalid time zone"
#     error message: 'value for domain timezone violates check constr
#     doesn't match: '.+"timezone_check"'
```

`lives_ok()`

```
SELECT lives_ok( :sql, :description );
SELECT lives_ok( :sql );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:description`

A short description of the test.

The inverse of `throws_ok()`, `lives_ok()` ensures that an SQL statement does *not* throw an exception. Pass in the name of a prepared statement or string of SQL code (see the summary for query argument details). The optional second argument is the test description. An example:


```
SELECT lives_ok(
    'INSERT INTO try (id) VALUES (1)',
    'We should not get a unique violation for a new PK'
);
```

A failing `lives_ok()` test produces an appropriate diagnostic message. For example:

```
# Failed test 85: "don't die, little buddy!"
#         died: 23505: duplicate key value violates unique constraint
```

Idea borrowed from the Test::Exception Perl module.

performs_ok()

```
SELECT performs_ok( :sql, :milliseconds, :description );
SELECT performs_ok( :sql, :milliseconds );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:milliseconds`

Number of milliseconds.

`:description`

A short description of the test.

This function makes sure that an SQL statement performs well. It does so by timing its execution and failing if execution takes longer than the specified number of milliseconds. An example:

```
PREPARE fast_query AS SELECT id FROM try WHERE name = 'Larry';
SELECT performs_ok(
    'fast_query',
    250,
    'A select by name should be fast'
);
```

The first argument should be the name of a prepared statement or a string representing the query to be executed (see the summary for query argument details). `performs_ok()` will use the PL/pgSQL `EXECUTE` statement to execute the query.

The second argument is the maximum number of milliseconds it should take for the SQL statement to execute. This argument is numeric, so you can even use fractions of milliseconds if it floats your boat.

The third argument is the usual description. If not provided, `performs_ok()` will generate the description "Should run in less than $milliseconds ms". You'll likely want to provide your own description if you have more than a couple of these in a test script or function.

Should a `performs_ok()` test fail it produces appropriate diagnostic messages. For example:

```
# Failed test 19: "The lookup should be fast!"
#       runtime: 200.266 ms
#       exceeds: 200 ms
```

Note: There is a little extra time included in the execution time for the the overhead of PL/pgSQL's `EXECUTE`, which must compile and execute the SQL string. You will want to account for this and pad your estimates accordingly. It's best to think of this as a brute force comparison of runtimes, in order to ensure that a query is not *really* slow (think seconds).

performs_within()

```
SELECT performs_within( :sql, :average_milliseconds, :within, :iterat
SELECT performs_within( :sql, :average_milliseconds, :within, :descri
SELECT performs_within( :sql, :average_milliseconds, :within, :iterat
SELECT performs_within( :sql, :average_milliseconds, :within);
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:average_milliseconds`

Number of milliseconds the query should take on average.

`:within`

The number of milliseconds that the average is allowed to vary.

`:iterations`

The number of times to run the query.

`:description`

A short description of the test.

This function makes sure that an SQL statement, on average, performs within an expected window. It does so by running the query a default of 10 times. It throws out the top and bottom 10% of runs, and averages the middle 80% of the runs it made. If the average execution time is outside the range specified by `within`, the test will fails. An example:

```
PREPARE fast_query AS SELECT id FROM try WHERE name = 'Larry';
SELECT performs_within(
    'fast_query',
    250,
    10,
    100,
    'A select by name should be fast'
);
```

The first argument should be the name of a prepared statement or a string representing the query to be executed (see the summary for query argument details). `performs_within()` will use the PL/pgSQL `EXECUTE` statement to execute the query.

The second argument is the average number of milliseconds it should take for the SQL statement to execute. This argument is numeric, so you can even use fractions of milliseconds if it floats your boat.

The third argument is the number of milliseconds the query is allowed to vary around the average and still still pass the test. If the query's average is falls outside this window, either too fast or too slow, it will fail.

The fourth argument is either the number of iterations or the usual description. If not provided, `performs_within()` will execute 10 runs of the query and will generate the description "Should run in $average_milliseconds +/- $within ms". You'll likely want to provide your own description if you have more than a couple of these in a test script or function.

The fifth argument is the usual description as described above, assuming you've also specified the number of iterations.

Should a `performs_within()` test fail it produces appropriate diagnostic messages. For example:

```
# Failed test 19: "The lookup should be fast!"
# average runtime: 210.266 ms
# desired average: 200 +/- 10 ms
```

Note: There is a little extra time included in the execution time for the the overhead of PL/pgSQL's `EXECUTE`, which must compile and execute the SQL string. You will want to account for this and pad your estimates accordingly. It's best to think of this as a brute force comparison of runtimes, in order to ensure that a query is not *really* slow (think seconds).

Can You Relate?

So you've got your basic scalar comparison functions, what about relations? Maybe you have some pretty hairy `SELECT` statements in views or functions to test? We've got your relation-testing functions right here.

`results_eq()`

```
SELECT results_eq( :sql,    :sql,    :description );
SELECT results_eq( :sql,    :sql                  );
SELECT results_eq( :sql,    :array,  :description );
SELECT results_eq( :sql,    :array                );
SELECT results_eq( :cursor, :cursor, :description );
SELECT results_eq( :cursor, :cursor               );
SELECT results_eq( :sql,    :cursor, :description );
SELECT results_eq( :sql,    :cursor               );
SELECT results_eq( :cursor, :sql,    :description );
SELECT results_eq( :cursor, :sql                  );
SELECT results_eq( :cursor, :array,  :description );
SELECT results_eq( :cursor, :array                );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:array`

An array of values representing a single-column row values.

`:cursor`

A PostgreSQL `refcursor` value representing a named cursor.

`:description`

A short description of the test.

There are three ways to test result sets in pgTAP. Perhaps the most intuitive is to do a direct row-by-row comparison of results to ensure that they are exactly what you expect, in the order you expect. Coincidentally, this is exactly how `results_eq()` behaves. Here's how you use it: simply pass in two SQL statements

or prepared statement names (or some combination; (see the summary for query argument details) and an optional description. Yep, that's it. It will do the rest.

For example, say that you have a function, `active_users()`, that returns a set of rows from the users table. To make sure that it returns the rows you expect, you might do something like this:

```
SELECT results_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active',
    'active_users() should return active users'
);
```

Tip: If you're using PostgreSQL 8.2 and up and want to hard-code the values to compare, use a `VALUES` statement instead of a query, like so:

```
SELECT results_eq(
    'SELECT * FROM active_users()',
    $$VALUES ( 42, 'Anna'), (19, 'Strongrrl'), (39, 'Theory')$$,
    'active_users() should return active users'
);
```

If the results returned by the first argument consist of a single column, the second argument may be an array:

```
SELECT results_eq(
    'SELECT * FROM active_user_ids()',
    ARRAY[ 2, 3, 4, 5]
);
```

In general, the use of prepared statements is highly recommended to keep your test code SQLish (you can even use `VALUES` in prepared statements in PostgreSQL 8.2 and up!). But note that, because `results_eq()` does a row-by-row comparison, the results of the two query arguments must be in exactly the same order, with exactly the same data types, in order to pass. In practical terms, it means that you must make sure that your results are never unambiguously ordered.

For example, say that you want to compare queries against a `persons` table. The simplest way to sort is by `name`, as in:

```
try=# select * from people order by name;
  name   | age
--------+-----
```


```
 Damian |  19
 Larry  |  53
 Tom    |  44
 Tom    |  35
(4 rows)
```

But a different run of the same query could have the rows in different order:

```
try=# select * from people order by name;
  name   | age
--------+-----
 Damian |  19
 Larry  |  53
 Tom    |  35
 Tom    |  44
(4 rows)
```

Notice how the two “Tom” rows are reversed. The upshot is that you must ensure that your queries are always fully ordered. In a case like the above, it means sorting on both the `name` column and the `age` column. If the sort order of your results isn’t important, consider using `set_eq()` or `bag_eq()` instead.

Internally, `results_eq()` turns your SQL statements into cursors so that it can iterate over them one row at a time. Conveniently, this behavior is directly available to you, too. Rather than pass in some arbitrary SQL statement or the name of a prepared statement, simply create a cursor and pass *it* in, like so:

```
DECLARE cwant CURSOR FOR SELECT * FROM active_users();
DECLARE chave CURSOR FOR SELECT * FROM users WHERE active ORDER BY na

SELECT results_eq(
    'cwant'::refcursor,
    'chave'::refcursor,
    'Gotta have those active users!'
);
```

The key is to ensure that the cursor names are passed as `refcursor`s. This allows `results_eq()` to disambiguate them from prepared statements. And of course, you can mix and match cursors, prepared statements, and SQL as much as you like. Here’s an example using a prepared statement and a (reset) cursor for the expected results:

```
PREPARE users_test AS SELECT * FROM active_users();
MOVE BACKWARD ALL IN chave;
```


```
SELECT results_eq(
    'users_test',
    'chave'::refcursor,
    'Gotta have those active users!'
);
```

Regardless of which types of arguments you pass, in the event of a test failure, `results_eq()` will offer a nice diagnostic message to tell you at what row the results differ, something like:

```
# Failed test 146
#     Results differ beginning at row 3:
#         have: (1,Anna)
#         want: (22,Betty)
```

If there are different numbers of rows in each result set, a non-existent row will be represented as "NULL":

```
# Failed test 147
#     Results differ beginning at row 5:
#         have: (1,Anna)
#         want: NULL
```

On PostgreSQL 8.4 or higher, if the number of columns varies between result sets, or if results are of different data types, you'll get diagnostics like so:

```
# Failed test 148
#     Number of columns or their types differ between the queries:
#         have: (1)
#         want: (foo,1)
```

On PostgreSQL 8.3 and down, the rows are cast to text for comparison, rather than compared as `record` objects. The downside to this necessity is that the test cannot detect incompatibilities in column numbers or types, or differences in columns that convert to the same text representation. For example, a `NULL` column will be equivalent to an empty string. As a result, pgTAP will not show the `have` and `want` values if they are the same, just the error message, like so:

```
# Failed test 149
#     Number of columns or their types differ between the queries
```

results_ne()

```
SELECT results_ne( :sql,    :sql,    :description );
SELECT results_ne( :sql,    :sql                  );
```


```
SELECT results_ne( :sql,    :array,  :description );
SELECT results_ne( :sql,    :array               );
SELECT results_ne( :cursor, :cursor, :description );
SELECT results_ne( :cursor, :cursor              );
SELECT results_ne( :sql,    :cursor, :description );
SELECT results_ne( :sql,    :cursor              );
SELECT results_ne( :cursor, :sql,    :description );
SELECT results_ne( :cursor, :sql                  );
SELECT results_ne( :cursor, :array,  :description );
SELECT results_ne( :cursor, :array               );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:array`

An array of values representing a single-column row values.

`:cursor`

A PostgreSQL `refcursor` value representing a named cursor.

`:description`

A short description of the test.

The inverse of `results_eq()`, this function tests that query results are not equivalent. Note that, like `results_ne()`, order matters, so you can actually have the same sets of results in the two query arguments and the test will pass if they're merely in a different order. More than likely what you really want is `results_eq()` or `set_ne()`. But this function is included for completeness and is kind of cute, so enjoy. If a `results_ne()` test fails, however, there will be no diagnostics, because, well, the results will be the same!

Note that the caveats for `results_ne()` on PostgreSQL 8.3 and down apply to `results_ne()` as well.

set_eq()

```
SELECT set_eq( :sql, :sql,   :description );
SELECT set_eq( :sql, :sql                 );
SELECT set_eq( :sql, :array, :description );
SELECT set_eq( :sql, :array               );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:array`

An array of values representing a single-column row values.

`:description`

A short description of the test.

Sometimes you don't care what order query results are in, or if there are duplicates. In those cases, use `set_eq()` to do a simple set comparison of your result sets. As long as both queries return the same records, regardless of duplicates or ordering, a `set_eq()` test will pass.

The SQL arguments can be the names of prepared statements or strings containing an SQL query (see the summary for query argument details), or even one of each. If the results returned by the first argument consist of a single column, the second argument may be an array:

```
SELECT set_eq(
    'SELECT * FROM active_user_ids()',
    ARRAY[ 2, 3, 4, 5]
);
```

In whatever case you choose to pass arguments, a failing test will yield useful diagnostics, such as:

```
# Failed test 146
#     Extra records:
#         (87,Jackson)
#         (1,Jacob)
#     Missing records:
#         (44,Anna)
#         (86,Angelina)
```

In the event that you somehow pass queries that return rows with different types of columns, pgTAP will tell you that, too:

```
# Failed test 147
#     Columns differ between queries:
#         have: (integer,text)
#         want: (text,integer)
```

This of course extends to sets with different numbers of columns:

```
# Failed test 148
#     Columns differ between queries:
```


```
#         have: (integer)
#         want: (text,integer)
```

set_ne()

```
SELECT set_ne( :sql, :sql,   :description );
SELECT set_ne( :sql, :sql                 );
SELECT set_ne( :sql, :array, :description );
SELECT set_ne( :sql, :array               );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:array`

An array of values representing a single-column row values.

`:description`

A short description of the test.

The inverse of `set_eq()`, this function tests that the results of two queries are *not* the same. The two queries can as usual be the names of prepared statements or strings containing an SQL query (see the summary for query argument details), or even one of each. The two queries, however, must return results that are directly comparable – that is, with the same number and types of columns in the same orders. If it happens that the query you're testing returns a single column, the second argument may be an array.

set_has()

```
SELECT set_has( :sql, :sql, :description );
SELECT set_has( :sql, :sql );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:description`

A short description of the test.

When you need to test that a query returns at least some subset of records, `set_has()` is the hammer you're looking for. It tests that the the results of a query contain at least the results returned by another query, if not more. That is, the test passes if the second query's results are a subset of the first query's

results. The second query can even return an empty set, in which case the test will pass no matter what the first query returns. Not very useful perhaps, but set-theoretically correct.

As with `set_eq()`. the SQL arguments can be the names of prepared statements or strings containing an SQL query (see the summary for query argument details), or one of each. If it happens that the query you're testing returns a single column, the second argument may be an array.

In whatever case, a failing test will yield useful diagnostics just like:

```
# Failed test 122
#     Missing records:
#         (44,Anna)
#         (86,Angelina)
```

As with `set_eq()`, `set_has()` will also provide useful diagnostics when the queries return incompatible columns. Internally, it uses an `EXCEPT` query to determine if there any any unexpectedly missing results.

set_hasnt()

```
SELECT set_hasnt( :sql, :sql, :description );
SELECT set_hasnt( :sql, :sql );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:description`

A short description of the test.

This test function is the inverse of `set_has()`: the test passes when the results of the first query have none of the results of the second query. Diagnostics are similarly useful:

```
# Failed test 198
#     Extra records:
#         (44,Anna)
#         (86,Angelina)
```

Internally, the function uses an `INTERSECT` query to determine if there is any unexpected overlap between the query results.

bag_eq()

```
SELECT bag_eq( :sql, :sql,   :description );
SELECT bag_eq( :sql, :sql                 );
SELECT bag_eq( :sql, :array, :description );
SELECT bag_eq( :sql, :array               );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:array`

An array of values representing a single-column row values.

`:description`

A short description of the test.

The `bag_eq()` function is just like `set_eq()`, except that it considers the results as bags rather than as sets. A bag is a set that allows duplicates. In practice, it mean that you can use `bag_eq()` to test result sets where order doesn't matter, but duplication does. In other words, if a two rows are the same in the first result set, the same row must appear twice in the second result set.

Otherwise, this function behaves exactly like `set_eq()`, including the utility of its diagnostics.

bag_ne()

```
SELECT bag_ne( :sql, :sql,   :description );
SELECT bag_ne( :sql, :sql                 );
SELECT bag_ne( :sql, :array, :description );
SELECT bag_ne( :sql, :array               );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:array`

An array of values representing a single-column row values.

`:description`

A short description of the test.

The inverse of `bag_eq()`, this function tests that the results of two queries are *not* the same, including duplicates. The two queries can as usual be the names of

prepared statements or strings containing an SQL query (see the summary for query argument details), or even one of each. The two queries, however, must return results that are directly comparable – that is, with the same number and types of columns in the same orders. If it happens that the query you're testing returns a single column, the second argument may be an array.

bag_has()

```
SELECT bag_has( :sql, :sql, :description );
SELECT bag_has( :sql, :sql );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:description`

A short description of the test.

The `bag_has()` function is just like `set_has()`, except that it considers the results as bags rather than as sets. A bag is a set with duplicates. What practice this means that you can use `bag_has()` to test result sets where order doesn't matter, but duplication does. Internally, it uses an `EXCEPT ALL` query to determine if there any any unexpectedly missing results.

bag_hasnt()

```
SELECT bag_hasnt( :sql, :sql, :description );
SELECT bag_hasnt( :sql, :sql );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:description`

A short description of the test.

This test function is the inverse of `bag_hasnt()`: the test passes when the results of the first query have none of the results of the second query. Diagnostics are similarly useful:

```
# Failed test 198
#     Extra records:
```


```
#         (44,Anna)
#         (86,Angelina)
```

Internally, the function uses an `INTERSECT ALL` query to determine if there is any unexpected overlap between the query results. This means that a duplicate row in the first query will appear twice in the diagnostics if it is also duplicated in the second query.

`is_empty()`

```
SELECT is_empty( :sql, :description );
SELECT is_empty( :sql );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:description`

A short description of the test.

The `is_empty()` function takes a single query string or prepared statement name as its first argument, and tests that said query returns no records. Internally it simply executes the query and if there are any results, the test fails and the results are displayed in the failure diagnostics, like so:

```
# Failed test 494: "Should have no inactive users"
#     Records returned:
#         (1,Jacob,false)
#         (2,Emily,false)
```

`isnt_empty()`

```
SELECT isnt_empty( :sql, :description );
SELECT isnt_empty( :sql );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:description`

A short description of the test.

This function is the inverse of `is_empty()`. The test passes if the specified query, when executed, returns at least one row. If it returns no rows, the test fails.

row_eq()

```
SELECT row_eq( :sql, :record, :description );
SELECT row_eq( :sql, :record );
```

Parameters

`:sql`

An SQL statement or the name of a prepared statement, passed as a string.

`:record`

A row or value, also known as a composite type.

`:description`

A short description of the test.

Compares the contents of a single row to a record. Due to the limitations of non-C functions in PostgreSQL, a bar `RECORD` value cannot be passed to the function. You must instead pass in a valid composite type value, and cast the record argument (the second argument) to the same type. Both explicitly created composite types and table types are supported. Thus, you can do this:

```
CREATE TYPE sometype AS (
    id    INT,
    name  TEXT
);

SELECT row_eq( $$ SELECT 1, 'foo' $$, ROW(1, 'foo')::sometype );
```

And, of course, this:

```
CREATE TABLE users (
    id   INT,
    name TEXT
);

INSERT INTO users VALUES (1, 'theory');
PREPARE get_user AS SELECT * FROM users LIMIT 1;

SELECT row_eq( 'get_user', ROW(1, 'theory')::users );
```

Compatible types can be compared, though. So if the `users` table actually included an `active` column, for example, and you only wanted to test the `id` and `name`, you could do this:

```
SELECT row_eq(
    $$ SELECT id, name FROM users $$,
```


```
    ROW(1, 'theory')::sometype
);
```

Note the use of the `sometype` composite type for the second argument. The upshot is that you can create composite types in your tests explicitly for comparing the return values of your queries, if such queries don't return an existing valid type.

Hopefully someday in the future we'll be able to support arbitrary `record` arguments. In the meantime, this is the 90% solution.

Diagnostics on failure are similar to those from `is()`:

```
# Failed test 322
#       have: (1,Jacob)
#       want: (1,Larry)
```

The Schema Things

Need to make sure that your database is designed just the way you think it should be? Use these test functions and rest easy.

A note on comparisons: pgTAP uses a simple equivalence test (=) to compare all SQL identifiers, such as the names of tables, schemas, functions, indexes, and columns (but not data types). So in general, you should always use lowercase strings when passing identifier arguments to the functions below. Use mixed case strings only when the objects were declared in your schema using double-quotes. For example, if you created a table like so:

```
CREATE TABLE Foo (id integer);
```

Then you *must* test for it using only lowercase characters (if you want the test to pass):

```
SELECT has_table('foo');
```

If, however, you declared the table using a double-quoted string, like so:

```
CREATE TABLE "Foo" (id integer);
```

Then you'd need to test for it using exactly the same string, including case, like so:


```
SELECT has_table('Foo');
```

In general, this should not be an issue, as mixed-case objects are created only rarely. So if you just stick to lowercase-only arguments to these functions, you should be in good shape.

I Object!

In a busy development environment, you might have a number of users who make changes to the database schema. Sometimes you have to really work to keep these folks in line. For example, do they add objects to the database without adding tests? Do they drop objects that they shouldn't? These assertions are designed to help you ensure that the objects in the database are exactly the objects that should be in the database, no more, no less.

Each tests tests that all of the objects in the database are only the objects that *should* be there. In other words, given a list of objects, say tables in a call to `tables_are()`, this assertion will fail if there are tables that are not in the list, or if there are tables in the list that are missing from the database. It can also be useful for testing replication and the success or failure of schema change deployments.

If you're more interested in the specifics of particular objects, skip to the next section.

`tablespaces_are()`

```
SELECT tablespaces_are( :tablespaces, :description );
SELECT tablespaces_are( :tablespaces );
```

Parameters

`:tablespaces`

An array of tablespace names.

`:description`

A short description of the test.

This function tests that all of the tablespaces in the database only the tablespaces that *should* be there. Example:

```
SELECT tablespaces_are(ARRAY[ 'dbspace', 'indexspace' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing tablespaces, like so:

```
# Failed test 121: "There should be the correct tablespaces"
#     Extra tablespaces:
#         trigspace
#     Missing tablespaces:
#         indexspace
```

schemas_are()

```
SELECT schemas_are( :schemas, :description );
SELECT schemas_are( :schemas );
```

Parameters

`:schemas`

An array of schema names.

`:description`

A short description of the test.

This function tests that all of the schemas in the database only the schemas that *should* be there, excluding system schemas and `information_schema`. Example:

```
SELECT schemas_are(ARRAY[ 'public', 'contrib', 'tap' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing schemas, like so:

```
# Failed test 106: "There should be the correct schemas"
#     Extra schemas:
#         __howdy__
#     Missing schemas:
#         someschema
```

tables_are()

```
SELECT tables_are( :schema, :tables, :description );
SELECT tables_are( :schema, :tables );
SELECT tables_are( :tables, :description );
SELECT tables_are( :tables );
```

Parameters

`:schema`

Name of a schema in which to find tables.

`:tables`

An array of table names.

`:description`

A short description of the test.

This function tests that all of the tables in the named schema, or that are visible in the search path, are only the tables that *should* be there. If the `:schema` argument is omitted, tables will be sought in the search path, excluding `pg_catalog` and `information_schema` If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT tables_are(
    'myschema',
    ARRAY[ 'users', 'widgets', 'gadgets', 'session' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing tables, like so:

```
# Failed test 91: "Schema public should have the correct tables"
#     Extra tables:
#         mallots
#         __test_table
#     Missing tables:
#         users
#         widgets
```

foreign_tables_are()

```
SELECT foreign_tables_are( :schema, :foreign_tables, :description );
SELECT foreign_tables_are( :schema, :foreign_tables );
SELECT foreign_tables_are( :foreign_tables, :description );
SELECT foreign_tables_are( :foreign_tables );
```

Parameters

`:schema`

Name of a schema in which to find foreign tables.

`:foreign_tables`

An array of foreign table names.

`:description`

A short description of the test.

This function tests that all of the foreign tables in the named schema, or that are visible in the search path, are only the foreign tables that *should* be there. If the `:schema` argument is omitted, foreign tables will be sought in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT foreign_tables_are(
    'myschema',
    ARRAY[ 'users', 'widgets', 'gadgets', 'session' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing foreign tables, like so:

```
# Failed test 91: "Schema public should have the correct foreign tabl
#     Extra foreign tables:
#         mallots
#         __test_table
#     Missing foreign tables:
#         users
#         widgets
```

views_are()

```
SELECT views_are( :schema, :views, :description );
SELECT views_are( :schema, :views );
SELECT views_are( :views, :description );
SELECT views_are( :views );
```

Parameters

`:schema`

Name of a schema in which to find views.

`:views`

An array of view names.

`:description`

A short description of the test.

This function tests that all of the views in the named schema, or that are visible in the search path, are only the views that *should* be there. If the `:schema` argument is omitted, views will be sought in the search path, excluding `pg_catalog` and `information_schema` If the description is omitted, a generally useful default description will be generated. Example:


```
SELECT views_are(
    'myschema',
    ARRAY[ 'users', 'widgets', 'gadgets', 'session' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing views, like so:

```
# Failed test 92: "Schema public should have the correct views"
#     Extra views:
#         v_userlog_tmp
#         __test_view
#     Missing views:
#         v_userlog
#         eated
```

materialized_views_are()

```
SELECT materialized_views_are( :schema, :materialized_views, :descrip
SELECT materialized_views_are( :schema, :materialized_views );
SELECT materialized_views_are( :materialized_views, :description );
SELECT materialized_views_are( :materialized_views );
```

Parameters

`:schema`

Name of a schema in which to find materialized views.

`:materialized_views`

An array of materialized view names.

`:description`

A short description of the test.

This function tests that all of the materialized views in the named schema, or that are visible in the search path, are only the materialized views that *should* be there. If the `:schema` argument is omitted, materialized views will be sought in the search path, excluding `pg_catalog` and `information_schema` If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT materialized_views_are(
    'myschema',
    ARRAY[ 'users', 'widgets', 'gadgets', 'session' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing materialized views, like so:

```
# Failed test 92: "Schema public should have the correct materialized
#     Extra materialized views:
#         v_userlog_tmp
#         __test_view
#     Missing materialized views:
#         v_userlog
#         eated
```

sequences_are()

```
SELECT sequences_are( :schema, :sequences, :description );
SELECT sequences_are( :schema, :sequences );
SELECT sequences_are( :sequences, :description );
SELECT sequences_are( :sequences );
```

Parameters

`:schema`

Name of a schema in which to find sequences.

`:sequences`

An array of sequence names.

`:description`

A short description of the test.

This function tests that all of the sequences in the named schema, or that are visible in the search path, are only the sequences that *should* be there. If the `:schema` argument is omitted, sequences will be sought in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT sequences_are(
    'myschema',
    ARRAY[ 'users', 'widgets', 'gadgets', 'session' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing sequences, like so:

```
# Failed test 93: "Schema public should have the correct sequences"
#     These are extra sequences:
#         seq_mallots
```


```
#         __test_table_seq
#     These sequences are missing:
#         users_seq
#         widgets_seq
```

columns_are()

```
SELECT columns_are( :schema, :table, :columns, :description );
SELECT columns_are( :schema, :table, :columns );
SELECT columns_are( :table,  :columns, :description );
SELECT columns_are( :table,  :columns );
```

Parameters

`:schema`

Name of a schema in which to find the `:table`.

`:table`

Name of a table in which to find columns.

`:columns`

An array of column names.

`:description`

A short description of the test.

This function tests that all of the columns on the named table are only the columns that *should* be on that table. If the `:schema` argument is omitted, the table must be visible in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT columns_are(
    'myschema',
    'atable',
    ARRAY[ 'id', 'name', 'rank', 'sn' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing columns, like so:

```
# Failed test 183: "Table users should have the correct columns"
#     Extra columns:
#         given_name
#         surname
#     Missing columns:
#         name
```

indexes_are()

```
SELECT indexes_are( :schema, :table, :indexes, :description );
SELECT indexes_are( :schema, :table, :indexes );
SELECT indexes_are( :table,  :indexes, :description );
SELECT indexes_are( :table,  :indexes );
```

Parameters

`:schema`

Name of a schema in which to find the `:table`.

`:table`

Name of a table in which to find indexes.

`:indexes`

An array of index names.

`:description`

A short description of the test.

This function tests that all of the indexes on the named table are only the indexes that *should* be on that table. If the `:schema` argument is omitted, the table must be visible in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT indexes_are(
    'myschema',
    'atable',
    ARRAY[ 'atable_pkey', 'idx_atable_name' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing indexes, like so:

```
# Failed test 180: "Table fou should have the correct indexes"
#     Extra indexes:
#         fou_pkey
#     Missing indexes:
#         idx_fou_name
```

triggers_are()

```
SELECT triggers_are( :schema, :table, :triggers, :description );
SELECT triggers_are( :schema, :table, :triggers );
SELECT triggers_are( :table,  :triggers, :description );
SELECT triggers_are( :table,  :triggers );
```

Parameters

`:schema`

Name of a schema in which to find the `:table`.

`:table`

Name of a table in which to find triggers.

`:triggers`

An array of trigger names.

`:description`

A short description of the test.

This function tests that all of the triggers on the named table are only the triggers that *should* be on that table. If the `:schema` argument is omitted, the table must be visible in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT triggers_are(
    'myschema',
    'atable',
    ARRAY[ 'atable_pkey', 'idx_atable_name' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing triggers, like so:

```
# Failed test 180: "Table fou should have the correct triggers"
#     Extra triggers:
#         set_user_pass
#     Missing triggers:
#         set_users_pass
```

`functions_are()`

```
SELECT functions_are( :schema, :functions, :description );
SELECT functions_are( :schema, :functions );
SELECT functions_are( :functions, :description );
SELECT functions_are( :functions );
```

Parameters

`:schema`

Name of a schema in which to find functions.

`:functions`

An array of function names.

`:description`

A short description of the test.

This function tests that all of the functions in the named schema, or that are visible in the search path, are only the functions that *should* be there. If the `:schema` argument is omitted, functions will be sought in the search path, excluding `pg_catalog` and `information_schema` If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT functions_are(
    'myschema',
    ARRAY[ 'foo', 'bar', 'frobnitz' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing functions, like so:

```
# Failed test 150: "Schema someschema should have the correct functio
#     Extra functions:
#         schnauzify
#     Missing functions:
#         frobnitz
```

roles_are()

```
SELECT roles_are( :roles, :description );
SELECT roles_are( :roles );
```

Parameters

`:roles`

An array of role names.

`:description`

A short description of the test.

This function tests that all of the roles in the database only the roles that *should* be there. Example:

```
SELECT roles_are(ARRAY[ 'postgres', 'someone', 'root' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing roles, like so:


```
# Failed test 195: "There should be the correct roles"
#     Extra roles:
#         root
#     Missing roles:
#         bobby
```

users_are()

```
SELECT users_are( :users, :description );
SELECT users_are( :users );
```

Parameters

`:users`

An array of user names.

`:description`

A short description of the test.

This function tests that all of the users in the database only the users that *should* be there. Example:

```
SELECT users_are(ARRAY[ 'postgres', 'someone', 'root' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing users, like so:

```
# Failed test 195: "There should be the correct users"
#     Extra users:
#         root
#     Missing users:
#         bobby
```

groups_are()

```
SELECT groups_are( :groups, :description );
SELECT groups_are( :groups );
```

Parameters

`:groups`

An array of group names.

`:description`

A short description of the test.

This function tests that all of the groups in the database only the groups that *should* be there. Example:


```
SELECT groups_are(ARRAY[ 'postgres', 'admins, 'l0s3rs' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing groups, like so:

```
# Failed test 210: "There should be the correct groups"
#     Extra groups:
#         meanies
#     Missing groups:
#         __howdy__
```

languages_are()

```
SELECT languages_are( :languages, :description );
SELECT languages_are( :languages );
```

Parameters

`:languages`

An array of language names.

`:description`

A short description of the test.

This function tests that all of the languages in the database only the languages that *should* be there. Example:

```
SELECT languages_are(ARRAY[ 'plpgsql', 'plperl', 'pllolcode' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing languages, like so:

```
# Failed test 225: "There should be the correct procedural languages"
#     Extra languages:
#         pllolcode
#     Missing languages:
#         plpgsql
```

opclasses_are()

```
SELECT opclasses_are( :schema, :opclasses, :description );
SELECT opclasses_are( :schema, :opclasses );
SELECT opclasses_are( :opclasses, :description );
SELECT opclasses_are( :opclasses );
```

Parameters

`:schema`

Name of a schema in which to find opclasses.

`:opclasses`

An array of opclass names.

`:description`

A short description of the test.

This function tests that all of the operator classes in the named schema, or that are visible in the search path, are only the opclasses that *should* be there. If the `:schema` argument is omitted, opclasses will be sought in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT opclasses_are(
    'myschema',
    ARRAY[ 'foo', 'bar', 'frobnitz' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing opclasses, like so:

```
# Failed test 251: "Schema public should have the correct operator cl
#     Extra operator classes:
#         goofy_ops
#     Missing operator classes:
#         custom_ops
```

rules_are()

```
SELECT rules_are( :schema, :table, :rules, :description );
SELECT rules_are( :schema, :table, :rules );
SELECT rules_are( :table,  :rules, :description );
SELECT rules_are( :table,  :rules );
```

Parameters

`:schema`

Name of a schema in which to find the `:table`.

`:table`

Name of a table in which to find rules.

`:rules`

An array of rule names.

`:description`

A short description of the test.

This function tests that all of the rules on the named relation are only the rules that *should* be on that relation (a table, view or a materialized view). If the `:schema` argument is omitted, the rules must be visible in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT rules_are(
    'myschema',
    'atable',
    ARRAY[ 'on_insert', 'on_update', 'on_delete' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing rules, like so:

```
# Failed test 281: "Relation public.users should have the correct rul
#     Extra rules:
#         on_select
#     Missing rules:
#         on_delete
```

types_are()

```
SELECT types_are( :schema, :types, :description );
SELECT types_are( :schema, :types );
SELECT types_are( :types, :description );
SELECT types_are( :types );
```

Parameters

`:schema`

Name of a schema in which to find types.

`:types`

An array of data type names.

`:description`

A short description of the test.

Tests that all of the types in the named schema are the only types in that schema, including base types, composite types, domains, enums, and pseudo-types. If the `:schema` argument is omitted, the types must be visible in the search path,

excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT types_are('myschema', ARRAY[ 'timezone', 'state' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing types, like so:

```
# Failed test 307: "Schema someschema should have the correct types"
#     Extra types:
#         sometype
#     Missing types:
#         timezone
```

domains_are()

```
SELECT domains_are( :schema, :domains, :description );
SELECT domains_are( :schema, :domains );
SELECT domains_are( :domains, :description );
SELECT domains_are( :domains );
```

Parameters

`:schema`

Name of a schema in which to find domains.

`:domains`

An array of data domain names.

`:description`

A short description of the test.

Tests that all of the domains in the named schema are the only domains in that schema. If the `:schema` argument is omitted, the domains must be visible in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT domains_are('myschema', ARRAY[ 'timezone', 'state' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing domains, like so:

```
# Failed test 327: "Schema someschema should have the correct domains
#     Extra domains:
#         somedomain
```


```
#     Missing domains:
#         timezone
```

enums_are()

```
SELECT enums_are( :schema, :enums, :description );
SELECT enums_are( :schema, :enums );
SELECT enums_are( :enums, :description );
SELECT enums_are( :enums );
```

Parameters

`:schema`

Name of a schema in which to find enums.

`:enums`

An array of enum data type names.

`:description`

A short description of the test.

Tests that all of the enums in the named schema are the only enums in that schema. Enums are supported in PostgreSQL 8.3 and up. If the `:schema` argument is omitted, the enums must be visible in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT enums_are('myschema', ARRAY[ 'timezone', 'state' ]);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing enums, like so:

```
# Failed test 333: "Schema someschema should have the correct enums"
#     Extra enums:
#         someenum
#     Missing enums:
#         bug_status
```

casts_are()

```
SELECT casts_are( :casts, :description );
SELECT casts_are( :casts );
```

Parameters

`:casts`

An array of cast names.

`:description`

A short description of the test.

This function tests that all of the casts in the database are only the casts that *should* be in that database. Casts are specified as strings in a syntax similarly to how they're declared via `CREATE CAST`. The pattern is `:source_type AS :target_type`. If either type was created with double-quotes to force mixed case or special characers, then you must use double quotes in the cast strings. Example:

```
SELECT casts_are(ARRAY[
    'integer AS "myInteger"',
    'integer AS double precision',
    'integer AS reltime',
    'integer AS numeric',
]);
```

If the description is omitted, a generally useful default description will be generated.

In the event of a failure, you'll see diagnostics listing the extra and/or missing casts, like so:

```
# Failed test 302: "There should be the correct casts"
#     Extra casts:
#         lseg AS point
#     Missing casts:
#         lseg AS integer
```

operators_are()

```
SELECT operators_are( :schema, :operators, :description );
SELECT operators_are( :schema, :operators );
SELECT operators_are( :operators, :description );
SELECT operators_are( :operators );
```

Parameters

`:schema`

Name of a schema in which to find operators.

`:operators`

An array of operators.

`:description`

A short description of the test.

Tests that all of the operators in the named schema are the only operators in that schema. If the `:schema` argument is omitted, the operators must be visible in the search path, excluding `pg_catalog` and `information_schema`. If the description is omitted, a generally useful default description will be generated.

The `:operators` argument is specified as an array of strings in which each operator is defined similarly to the display of the `:regoperator` type. The format is `:op(:leftop,:rightop) RETURNS :return_type`.

For left operators the left argument type should be `NONE`. For right operators, the right argument type should be `NONE`. The example above shows one one of each of the operator types. `=(citext,citext)` is an infix operator, `-(bigint,NONE)` is a left operator, and `!(NONE,bigint)` is a right operator. Example:

```
SELECT operators_are(
    'public',
    ARRAY[
        '=(citext,citext) RETURNS boolean',
        '-(NONE,bigint) RETURNS bigint',
        '!(bigint,NONE) RETURNS numeric'
    ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing operators, like so:

```
# Failed test 453: "Schema public should have the correct operators"
#     Extra operators:
#         +(integer,integer) RETURNS integer
#     Missing enums:
#         +(integer,text) RETURNS text
```

extensions_are()

```
SELECT extensions_are( :schema, :extensions, :description );
SELECT extensions_are( :schema, :extensions );
SELECT extensions_are( :extensions, :description );
SELECT extensions_are( :extensions );
```

Parameters

`:schema`

Name of a schema in which to find extensions.

`:extensions`

An array of extension names.

`:description`

A short description of the test.

This function tests that all of the extensions in the named schema, or that are visible in the search path, are only the extensions that *should* be there. If the `:schema` argument is omitted, all extensions will be checked. If the description is omitted, a generally useful default description will be generated. Example:

```
SELECT extensions_are(
    'myschema',
    ARRAY[ 'citext', 'isn', 'plpgsql' ]
);
```

In the event of a failure, you'll see diagnostics listing the extra and/or missing extensions, like so:

```
# Failed test 91: "Schema public should have the correct extensions"
#     Extra extensions:
#         pgtap
#         ltree
#     Missing extensions:
#         citext
#         isn
```

To Have or Have Not

Perhaps you're not so concerned with ensuring the precise correlation of database objects. Perhaps you just need to make sure that certain objects exist (or that certain objects *don't* exist). You've come to the right place.

has_tablespace()

```
SELECT has_tablespace( :tablespace, :location, :description );
SELECT has_tablespace( :tablespace, :description );
SELECT has_tablespace( :tablespace );
```

Parameters

`:tablespace`

Name of a tablespace.

`:location`

The tablespace's Location on disk.

`:description`

A short description of the test.

This function tests whether or not a tablespace exists in the database. The first argument is a tablespace name. The second is either the a file system path for the database or a test description. If you specify a location path, you must pass a description as the third argument; otherwise, if you omit the test description, it will be set to "Tablespace `:tablespace` should exist". Example:

```
SELECT has_tablespace('sometablespace', '/data/dbs');
```

hasnt_tablespace()

```
SELECT hasnt_tablespace( :tablespace, :description );
SELECT hasnt_tablespace( :tablespace );
```

Parameters

`:tablespace`

Name of a tablespace.

`:description`

A short description of the test.

This function is the inverse of `has_tablespace()`. The test passes if the specified tablespace does *not* exist.

has_schema()

```
SELECT has_schema( :schema, :description );
SELECT has_schema( :schema );
```

Parameters

`:schema`

Name of a schema.

`:description`

A short description of the test.

This function tests whether or not a schema exists in the database. The first argument is a schema name and the second is the test description. If you omit the test description, it will be set to "Schema `:schema` should exist".

hasnt_schema()


```
SELECT hasnt_schema(
    'someschema',
    'There should be no schema someschema'
);
```

Parameters

`:schema`

Name of a schema.

`:description`

A short description of the test.

This function is the inverse of `has_schema()`. The test passes if the specified schema does *not* exist.

has_relation()

```
SELECT has_relation( :schema, :relation, :description );
SELECT has_relation( :relation, :description );
SELECT has_relation( :relation );
```

Parameters

`:schema`

Name of a schema in which to find the relation.

`:relation`

Name of a relation.

`:description`

A short description of the test.

This function tests whether or not a relation exists in the database. Relations are tables, views, materialized views, seqences, composite types, foreign tables, and toast tables. The first argument is a schema name, the second is a relation name, and the third is the test description. If you omit the schema, the relation must be visible in the search path. Example:

```
SELECT has_relation('myschema', 'somerelation');
```

If you omit the test description, it will be set to “Relation `:relation` should exist”.

hasnt_relation()


```
SELECT hasnt_relation( :schema, :relation, :description );
SELECT hasnt_relation( :relation, :description );
SELECT hasnt_relation( :relation );
```

Parameters

`:schema`

Name of a schema in which to find the relation.

`:relation`

Name of a relation.

`:description`

A short description of the test.

This function is the inverse of `has_relation()`. The test passes if the specified relation does *not* exist.

has_table()

```
SELECT has_table( :schema, :table, :description );
SELECT has_table( :schema, :table );
SELECT has_table( :table, :description );
SELECT has_table( :table );
```

Parameters

`:schema`

Name of a schema in which to find the table.

`:table`

Name of a table.

`:description`

A short description of the test.

This function tests whether or not a table exists in the database. The first argument is a schema name, the second is a table name, and the third is the test description. If you omit the schema, the table must be visible in the search path. Example:

```
SELECT has_table('myschema'::name, 'sometable'::name);
```

If you omit the test description, it will be set to "Table `:table` should exist".

Note that this function will not recognize foreign tables; use `has_foreign_table()` to test for the presence of foreign tables.

hasnt_table()

```
SELECT hasnt_table( :schema, :table, :description );
SELECT hasnt_table( :schema, :table );
SELECT hasnt_table( :table, :description );
SELECT hasnt_table( :table );
```

Parameters

`:schema`

Name of a schema in which to find the table.

`:table`

Name of a table.

`:description`

A short description of the test.

This function is the inverse of `has_table()`. The test passes if the specified table does *not* exist.

has_view()

```
SELECT has_view( :schema, :view, :description );
SELECT has_view( :view, :description );
SELECT has_view( :view );
```

Parameters

`:schema`

Name of a schema in which to find the view.

`:view`

Name of a view.

`:description`

A short description of the test.

This function tests whether or not a view exists in the database. The first argument is a schema name, the second is a view name, and the third is the test description. If you omit the schema, the view must be visible in the search path. Example:

```
SELECT has_view('myschema', 'someview');
```

If you omit the test description, it will be set to “View `:view` should exist”.

hasnt_view()


```
SELECT hasnt_view( :schema, :view, :description );
SELECT hasnt_view( :view, :description );
SELECT hasnt_view( :view );
```

Parameters

`:schema`

Name of a schema in which to find the view.

`:view`

Name of a view.

`:description`

A short description of the test.

This function is the inverse of `has_view()`. The test passes if the specified view does *not* exist.

has_materialized_view()

```
SELECT has_materialized_view( :schema, :materialized_view, :descripti
SELECT has_materialized_view( :materialized_view, :description );
SELECT has_materialized_view( :materialized_view );
```

Parameters

`:schema`

Name of a schema in which to find the materialized view.

`:materialized_view`

Name of a materialized view.

`:description`

A short description of the test.

This function tests whether or not a materialized view exists in the database. The first argument is a schema name, the second is a materialized view name, and the third is the test description. If you omit the schema, the materialized view must be visible in the search path. Example:

```
SELECT has_materialized_view('myschema', 'some_materialized_view');
```

If you omit the test description, it will be set to “Materialized view `:materialized_view` should exist”.

hasnt_materialized_view()


```
SELECT hasnt_materialized_view( :schema, :materialized_view, :descrip
SELECT hasnt_materialized_view( :materialized_view, :description );
SELECT hasnt_materialized_view( :materialized_view );
```

Parameters

`:schema`

Name of a schema in which to find the materialized view.

`:materialized_view`

Name of a materialized view.

`:description`

A short description of the test.

This function is the inverse of `has_view()`. The test passes if the specified materialized view does *not* exist.

has_sequence()

```
SELECT has_sequence( :schema, :sequence, :description );
SELECT has_sequence( :schema, :sequence );
SELECT has_sequence( :sequence, :description );
SELECT has_sequence( :sequence );
```

Parameters

`:schema`

Name of a schema in which to find the sequence.

`:sequence`

Name of a sequence.

`:description`

A short description of the test.

This function tests whether or not a sequence exists in the database. The first argument is a schema name, the second is a sequence name, and the third is the test description. If you omit the schema, the sequence must be visible in the search path. Example:

```
SELECT has_sequence('somesequence');
```

If you omit the test description, it will be set to "Sequence `:schema.:sequence` should exist". If you find that the function call seems to be getting confused, cast the sequence to the `NAME` type:


```
SELECT has_sequence('myschema', 'somesequence'::NAME);
```

hasnt_sequence()

```
SELECT hasnt_sequence( :schema, :sequence, :description );
SELECT hasnt_sequence( :sequence, :description );
SELECT hasnt_sequence( :sequence );
```

Parameters

`:schema`

Name of a schema in which to find the sequence.

`:sequence`

Name of a sequence.

`:description`

A short description of the test.

This function is the inverse of `has_sequence()`. The test passes if the specified sequence does *not* exist.

has_foreign_table()

```
SELECT has_foreign_table( :schema, :table, :description );
SELECT has_foreign_table( :schema, :table );
SELECT has_foreign_table( :table, :description );
SELECT has_foreign_table( :table );
```

Parameters

`:schema`

Name of a schema in which to find the foreign table.

`:table`

Name of a foreign table.

`:description`

A short description of the test.

This function tests whether or not a foreign table exists in the database. The first argument is a schema name, the second is a foreign table name, and the third is the test description. If you omit the schema, the foreign table must be visible in the search path. Example:

```
SELECT has_foreign_table('myschema'::name, 'some_foreign_table'::name
```

If you omit the test description, it will be set to “Foreign table `:table` should exist”.

`hasnt_foreign_table()`

```
SELECT hasnt_foreign_table( :schema, :table, :description );
SELECT hasnt_foreign_table( :schema, :table );
SELECT hasnt_foreign_table( :table, :description );
SELECT hasnt_foreign_table( :table );
```

Parameters

`:schema`

Name of a schema in which to find the foreign table.

`:table`

Name of a foreign table.

`:description`

A short description of the test.

This function is the inverse of `has_foreign_table()`. The test passes if the specified foreign table does *not* exist.

`has_type()`

```
SELECT has_type( schema, type, description );
SELECT has_type( schema, type );
SELECT has_type( type, description );
SELECT has_type( type );
```

Parameters

`:schema`

Name of a schema in which to find the data type.

`:type`

Name of a data type.

`:description`

A short description of the test.

This function tests whether or not a type exists in the database. Detects all types of types, including base types, composite types, domains, enums, and pseudo-types. The first argument is a schema name, the second is a type name, and the third is the test description. If you omit the schema, the type must be visible in the search path. If you omit the test description, it will be set to “Type `:type`

should exist". If you're passing a schema and type rather than type and description, be sure to cast the arguments to `name` values so that your type name doesn't get treated as a description. Example:

```
SELECT has_type( 'myschema', 'sometype' );
```

If you've created a composite type and want to test that the composed types are a part of it, use the column testing functions to verify them, like so:

```
CREATE TYPE foo AS (id int, name text);
SELECT has_type( 'foo' );
SELECT has_column( 'foo', 'id' );
SELECT col_type_is( 'foo', 'id', 'integer' );
```

hasnt_type()

```
SELECT hasnt_type( schema, type, description );
SELECT hasnt_type( schema, type );
SELECT hasnt_type( type, description );
SELECT hasnt_type( type );
```

Parameters

`:schema`

Name of a schema in which to find the data type.

`:type`

Name of a data type.

`:description`

A short description of the test.

This function is the inverse of `has_type()`. The test passes if the specified type does *not* exist.

has_composite()

```
SELECT has_composite( schema, type, description );
SELECT has_composite( schema, type );
SELECT has_composite( type, description );
SELECT has_composite( type );
```

Parameters

`:schema`

Name of a schema in which to find the composite type.

`:composite type`

Name of a composite type.

`:description`

A short description of the test.

This function tests whether or not a composite type exists in the database. The first argument is a schema name, the second is the name of a composite type, and the third is the test description. If you omit the schema, the composite type must be visible in the search path. If you omit the test description, it will be set to "Composite type `:composite type` should exist". Example:

```
SELECT has_composite( 'myschema', 'somecomposite' );
```

If you're passing a schema and composite type rather than composite type and description, be sure to cast the arguments to `name` values so that your composite type name doesn't get treated as a description.

`hasnt_composite()`

```
SELECT hasnt_composite( schema, type, description );
SELECT hasnt_composite( schema, type );
SELECT hasnt_composite( type, description );
SELECT hasnt_composite( type );
```

Parameters

`:schema`

Name of a schema in which to find the composite type.

`:composite type`

Name of a composite type.

`:description`

A short description of the test.

This function is the inverse of `has_composite()`. The test passes if the specified composite type does *not* exist.

`has_domain()`

```
SELECT has_domain( schema, domain, description );
SELECT has_domain( schema, domain );
SELECT has_domain( domain, description );
SELECT has_domain( domain );
```

Parameters

`:schema`

Name of a schema in which to find the domain.

`:domain`

Name of a domain.

`:description`

A short description of the test.

This function tests whether or not a domain exists in the database. The first argument is a schema name, the second is the name of a domain, and the third is the test description. If you omit the schema, the domain must be visible in the search path. If you omit the test description, it will be set to "Domain `:domain` should exist". Example:

```
SELECT has_domain( 'myschema', 'somedomain' );
```

If you're passing a schema and domain rather than domain and description, be sure to cast the arguments to `name` values so that your domain name doesn't get treated as a description.

`hasnt_domain()`

```
SELECT hasnt_domain( schema, domain, description );
SELECT hasnt_domain( schema, domain );
SELECT hasnt_domain( domain, description );
SELECT hasnt_domain( domain );
```

Parameters

`:schema`

Name of a schema in which to find the domain.

`:domain`

Name of a domain.

`:description`

A short description of the test.

This function is the inverse of `has_domain()`. The test passes if the specified domain does *not* exist.

`has_enum()`

```
SELECT has_enum( schema, enum, description );
SELECT has_enum( schema, enum );
```


```
SELECT has_enum( enum, description );
SELECT has_enum( enum );
```

Parameters

`:schema`

Name of a schema in which to find the enum.

`:enum`

Name of a enum.

`:description`

A short description of the test.

This function tests whether or not a enum exists in the database. Enums are supported in PostgreSQL 8.3 or higher. The first argument is a schema name, the second is the an enum name, and the third is the test description. If you omit the schema, the enum must be visible in the search path. If you omit the test description, it will be set to "Enum `:enum` should exist". Example:

```
SELECT has_enum( 'myschema', 'someenum' );
```

If you're passing a schema and enum rather than enum and description, be sure to cast the arguments to `name` values so that your enum name doesn't get treated as a description.

`hasnt_enum()`

```
SELECT hasnt_enum( schema, enum, description );
SELECT hasnt_enum( schema, enum );
SELECT hasnt_enum( enum, description );
SELECT hasnt_enum( enum );
```

Parameters

`:schema`

Name of a schema in which to find the enum.

`:enum`

Name of a enum.

`:description`

A short description of the test.

This function is the inverse of `has_enum()`. The test passes if the specified enum does *not* exist.

has_index()

```
SELECT has_index( :schema, :table, :index, :columns, :description );
SELECT has_index( :schema, :table, :index, :columns );
SELECT has_index( :schema, :table, :index, :column, :description );
SELECT has_index( :schema, :table, :index, :column );
SELECT has_index( :table,  :index, :columns, :description );
SELECT has_index( :table,  :index, :columns, :description );
SELECT has_index( :table,  :index, :column, :description );
SELECT has_index( :schema, :table, :index, :column );
SELECT has_index( :table,  :index, :column );
SELECT has_index( :schema, :table, :index );
SELECT has_index( :table,  :index, :description );
SELECT has_index( :table,  :index );
```

Parameters

`:schema`

Name of a schema in which to find the index.

`:table`

Name of a table in which to find index.

`:index`

Name of an index.

`:columns`

Array of the columns and/or expressions in the index.

`:column`

Idexed column name or expression.

`:description`

A short description of the test.

Checks for the existence of an index associated with the named table. The `:schema` argument is optional, as is the column name or names or expression, and the description. The columns argument may be a string naming one column or expression, or an array of column names and/or expressions. For expressions, you must use lowercase for all SQL keywords and functions to properly compare to PostgreSQL's internal form of the expression. Non-functional expressions should also be wrapped in parentheses. A few examples:

```
SELECT has_index(
    'myschema',
    'sometable',
    'myindex',
    ARRAY[ 'somecolumn', 'anothercolumn', 'lower(txtcolumn)' ],
```


```
    'Index "myindex" should exist'
);

SELECT has_index('myschema', 'sometable', 'anidx', 'somecolumn');
SELECT has_index('myschema', 'sometable', 'loweridx', '(somearray[1])');
SELECT has_index('sometable', 'someindex');
```

If you find that the function call seems to be getting confused, cast the index name to the `NAME` type:

```
SELECT has_index( 'public', 'sometab', 'idx_foo', 'name'::name );
```

If the index does not exist, `has_index()` will output a diagnostic message such as:

```
# Index "blah" ON public.sometab not found
```

If the index was found but the column specification or expression is incorrect, the diagnostics will look more like this:

```
#       have: "idx_baz" ON public.sometab(lower(name))
#       want: "idx_baz" ON public.sometab(lower(lname))
```

hasnt_index()

```
SELECT hasnt_index( schema, table, index, description );
SELECT hasnt_index( schema, table, index );
SELECT hasnt_index( table, index, description );
SELECT hasnt_index( table, index );
```

Parameters

`:schema`

Name of a schema in which to not find the index.

`:table`

Name of a table in which to not find the index.

`:index`

Name of an index.

`:description`

A short description of the test.

This function is the inverse of `has_index()`. The test passes if the specified index does *not* exist.

has_trigger()


```
SELECT has_trigger( :schema, :table, :trigger, :description );
SELECT has_trigger( :schema, :table, :trigger );
SELECT has_trigger( :table, :trigger, :description );
SELECT has_trigger( :table, :trigger )` ###
```

Parameters

`:schema`

Name of a schema in which to find the trigger.

`:table`

Name of a table in which to find the trigger.

`:trigger`

Name of an trigger.

`:description`

A short description of the test.

Tests to see if the specified table has the named trigger. The `:description` is optional, and if the schema is omitted, the table with which the trigger is associated must be visible in the search path.

`hasnt_trigger()`

```
SELECT hasnt_trigger( :schema, :table, :trigger, :description );
SELECT hasnt_trigger( :schema, :table, :trigger );
SELECT hasnt_trigger( :table, :trigger, :description );
SELECT hasnt_trigger( :table, :trigger )` ###
```

Parameters

`:schema`

Name of a schema in which to not find the trigger.

`:table`

Name of a table in which to not find the trigger.

`:trigger`

Name of an trigger.

`:description`

A short description of the test.

This function is the inverse of `has_trigger()`. The test passes if the specified trigger does *not* exist.

`has_rule()`


```
SELECT has_rule( :schema, :table, :rule, :description );
SELECT has_rule( :schema, :table, :rule );
SELECT has_rule( :table, :rule, :description );
SELECT has_rule( :table, :rule )` ###
```

Parameters

`:schema`

Name of a schema in which to find the rule.

`:table`

Name of a table in which to find the rule.

`:rule`

Name of an rule.

`:description`

A short description of the test.

Tests to see if the specified table has the named rule. The `:description` is optional, and if the schema is omitted, the table with which the rule is associated must be visible in the search path.

`hasnt_rule()`

```
SELECT hasnt_rule( :schema, :table, :rule, :description );
SELECT hasnt_rule( :schema, :table, :rule );
SELECT hasnt_rule( :table, :rule, :description );
SELECT hasnt_rule( :table, :rule )` ###
```

Parameters

`:schema`

Name of a schema in which to not find the rule.

`:table`

Name of a table in which to not find the rule.

`:rule`

Name of an rule.

`:description`

A short description of the test.

This function is the inverse of `has_rule()`. The test passes if the specified rule does *not* exist.

`has_function()`


```
SELECT has_function( :schema, :function, :args, :description );
SELECT has_function( :schema, :function, :args );
SELECT has_function( :schema, :function, :description );
SELECT has_function( :schema, :function );
SELECT has_function( :function, :args, :description );
SELECT has_function( :function, :args );
SELECT has_function( :function, :description );
SELECT has_function( :function );
```

Parameters

`:schema`

Name of a schema in which to not find the function.

`:function`

Name of a function.

`:args`

Array of data types of the function arguments.

`:description`

A short description of the test.

Checks to be sure that the given function exists in the named schema and with the specified argument data types. If `:schema` is omitted, `has_function()` will search for the function in the schemas defined in the search path. If `:args` is omitted, `has_function()` will see if the function exists without regard to its arguments. Some examples:

```
SELECT has_function(
    'pg_catalog',
    'decode',
    ARRAY[ 'text', 'text' ],
    'Function decode(text, text) should exist'
);

SELECT has_function( 'do_something' );
SELECT has_function( 'do_something', ARRAY['integer'] );
SELECT has_function( 'do_something', ARRAY['numeric'] );
```

The `:args` argument should be formatted as it would be displayed in the view of a function using the `\df` command in `psql`. For example, even if you have a numeric column with a precision of 8, you should specify `ARRAY['numeric']`. If you created a `varchar(64)` column, you should pass the `:args` argument as `ARRAY['character varying']`.

If you wish to use the two-argument form of `has_function()`, specifying only the schema and the function name, you must cast the `:function` argument to `:name` in order to disambiguate it from from the `has_function(:function, :description)` form. If you neglect to do so, your results will be unexpected.

Also, if you use the string form to specify the `:args` array, be sure to cast it to `name` to disambiguate it from a text string:

```
SELECT has_function( 'lower', '{text}'::name[] );
```

Deprecation notice: The old name for this test function, `can_ok()`, is still available, but emits a warning when called. It will be removed in a future version of pgTAP.

hasnt_function()

```
SELECT hasnt_function( :schema, :function, :args, :description );
SELECT hasnt_function( :schema, :function, :args );
SELECT hasnt_function( :schema, :function, :description );
SELECT hasnt_function( :schema, :function );
SELECT hasnt_function( :function, :args, :description );
SELECT hasnt_function( :function, :args );
SELECT hasnt_function( :function, :description );
SELECT hasnt_function( :function );
```

Parameters

`:schema`

Name of a schema in which not to find the function.

`:function`

Name of a function.

`:args`

Array of data types of the function arguments.

`:description`

A short description of the test.

This function is the inverse of `has_function()`. The test passes if the specified function (optionally with the specified signature) does *not* exist.

has_cast()

```
SELECT has_cast( :source_type, :target_type, :schema, :function, :des
SELECT has_cast( :source_type, :target_type, :schema, :function );
SELECT has_cast( :source_type, :target_type, :function, :description
```


```
SELECT has_cast( :source_type, :target_type, :function );
SELECT has_cast( :source_type, :target_type, :description );
SELECT has_cast( :source_type, :target_type );
```

Parameters

`:source_type`

Data type of the source value without typemod.

`:target_type`

Data type of the target value without typemod.

`:schema`

Schema in which to find the operator function.

`:function`

Name of the operator function.

`:description`

A short description of the test.

Tests for the existence of a cast. A cast consists of a source data type, a target data type, and perhaps a (possibly schema-qualified) function. An example:

```
SELECT has_cast( 'integer', 'bigint', 'pg_catalog', 'int8' );
```

If you omit the description for the 3- or 4-argument version, you'll need to cast the function name to the `NAME` data type so that PostgreSQL doesn't resolve the function name as a description. For example:

```
SELECT has_cast( 'integer', 'bigint', 'int8'::NAME );
```

pgTAP will generate a useful description if you don't provide one.

Note that pgTAP does not compare typemods. So if you wanted to test for a cast between, say, a `uuid` type and `bit(128)`, this will not work:

```
SELECT has_cast( 'integer', 'bit(128)' );
```

But this will:

```
SELECT has_cast( 'integer', 'bit' );
```

hasnt_cast()

```
SELECT hasnt_cast( :source_type, :target_type, :schema, :function, :d
SELECT hasnt_cast( :source_type, :target_type, :schema, :function );
```


```
SELECT hasnt_cast( :source_type, :target_type, :function, :descriptio
SELECT hasnt_cast( :source_type, :target_type, :function );
SELECT hasnt_cast( :source_type, :target_type, :description );
SELECT hasnt_cast( :source_type, :target_type );
```

Parameters

`:source_type`

Data type of the source value.

`:target_type`

Data type of the target value.

`:schema`

Schema in which not to find the operator function.

`:function`

Name of the operator function.

`:description`

A short description of the test.

This function is the inverse of `has_cast()`: the test passes if the specified cast does *not* exist.

has_operator()

```
SELECT has_operator( :left_type, :schema, :name, :right_type, :return
SELECT has_operator( :left_type, :schema, :name, :right_type, :return
SELECT has_operator( :left_type, :name, :right_type, :return_type, :d
SELECT has_operator( :left_type, :name, :right_type, :return_type );
SELECT has_operator( :left_type, :name, :right_type, :description );
SELECT has_operator( :left_type, :name, :right_type );
```

Parameters

`:left_type`

Data type of the left operand.

`:schema`

Schema in which to find the operator.

`:name`

Name of the operator.

`:right_type`

Data type of the right operand.

`:return_type`

Data type of the return value.

`:description`

A short description of the test.

Tests for the presence of a binary operator. If the operator exists with the given schema, name, left and right arguments, and return value, the test will fail. If the operator does not exist, the test will fail. Example:

```
SELECT has_operator( 'integer', 'pg_catalog', '<=', 'integer', 'boole
```

If you omit the schema name, then the operator must be visible in the search path. If you omit the test description, pgTAP will generate a reasonable one for you. The return value is also optional. If you need to test for a left or right unary operator, use `has_leftop()` or `has_rightop()` instead.

`has_leftop()`

```
SELECT has_leftop( :schema, :name, :type, :return_type, :description
SELECT has_leftop( :schema, :name, :type, :return_type );
SELECT has_leftop( :name, :type, :return_type, :description );
SELECT has_leftop( :name, :type, :return_type );
SELECT has_leftop( :name, :type, :description );
SELECT has_leftop( :name, :type );
```

Parameters

`:schema`

Schema in which to find the operator.

`:name`

Name of the operator.

`:type`

Data type of the operand.

`:return_type`

Data type of the return value.

`:description`

A short description of the test.

Tests for the presence of a left-unary operator. If the operator exists with the given schema, name, right argument, and return value, the test will fail. If the operator does not exist, the test will fail. Example:


```
SELECT has_leftop( 'pg_catalog', '!!', 'bigint', 'numeric' );
```

If you omit the schema name, then the operator must be visible in the search path. If you omit the test description, pgTAP will generate a reasonable one for you. The return type is also optional.

has_rightop()

```
SELECT has_rightop( :schema, :name, :type, :return_type, :description
SELECT has_rightop( :schema, :name, :type, :return_type );
SELECT has_rightop( :name, :type, :return_type, :description );
SELECT has_rightop( :name, :type, :return_type );
SELECT has_rightop( :name, :type, :description );
SELECT has_rightop( :name, :type );
```

Parameters

`:schema`
: Schema in which to find the operator.

`:name`
: Name of the operator.

`:type`
: Data type of the operand.

`:return_type`
: Data type of the return value.

`:description`
: A short description of the test.

Tests for the presence of a right-unary operator. If the operator exists with the given left argument, schema, name, and return value, the test will fail. If the operator does not exist, the test will fail. Example:

```
SELECT has_rightop( 'bigint', 'pg_catalog', '!', 'numeric' );
```

If you omit the schema name, then the operator must be visible in the search path. If you omit the test description, pgTAP will generate a reasonable one for you. The return type is also optional.

has_opclass()

```
SELECT has_opclass( :schema, :name, :description );
SELECT has_opclass( :schema, :name );
```


```
SELECT has_opclass( :name, :description );
SELECT has_opclass( :name );
```

Parameters

`:schema`

Schema in which to find the operator class.

`:name`

Name of the operator class.

`:description`

A short description of the test.

Tests for the presence of an operator class. If you omit the schema name, then the operator must be visible in the search path. If you omit the test description, pgTAP will generate a reasonable one for you. The return value is also optional.

`hasnt_opclass()`

```
SELECT hasnt_opclass( :schema, :name, :description );
SELECT hasnt_opclass( :schema, :name );
SELECT hasnt_opclass( :name, :description );
SELECT hasnt_opclass( :name );
```

Parameters

`:schema`

Schema in which not to find the operator class.

`:name`

Name of the operator class.

`:description`

A short description of the test.

This function is the inverse of `has_opclass()`. The test passes if the specified operator class does *not* exist.

`has_role()`

```
SELECT has_role( :role, :description );
SELECT has_role( :role );
```

Parameters

`:role`

Name of the role.

`:description`

A short description of the test.

Checks to ensure that a database role exists. If the description is omitted, it will default to "Role `:role` should exist".

`hasnt_role()`

```
SELECT hasnt_role( :role, :description );
SELECT hasnt_role( :role );
```

Parameters

`:role`

Name of the role.

`:description`

A short description of the test.

The inverse of `has_role()`, this function tests for the *absence* of a database role.

`has_user()`

```
SELECT has_user( :user, :description );
SELECT has_user( :user );
```

Parameters

`:user`

Name of the user.

`:description`

A short description of the test.

Checks to ensure that a database user exists. If the description is omitted, it will default to "User `:user` should exist".

`hasnt_user()`

```
SELECT hasnt_user( :user, :description );
SELECT hasnt_user( :user );
```

Parameters

`:user`

Name of the user.

`:description`

A short description of the test.

The inverse of `has_user()`, this function tests for the *absence* of a database user.

has_group()

```
SELECT has_group( :group, :description );
SELECT has_group( :group );
```

Parameters

`:group`

Name of the group.

`:description`

A short description of the test.

Checks to ensure that a database group exists. If the description is omitted, it will default to "Group `:group` should exist".

hasnt_group()

```
SELECT hasnt_group( :group, :description );
SELECT hasnt_group( :group );
```

Parameters

`:group`

Name of the group.

`:description`

A short description of the test.

The inverse of `has_group()`, this function tests for the *absence* of a database group.

has_language()

```
SELECT has_language( :language, :description );
SELECT has_language( :language );
```

Parameters

`:language`

Name of the language.

`:description`

A short description of the test.

Checks to ensure that a procedural language exists. If the description is omitted, it will default to “Procedural language `:language` should exist”.

`hasnt_language()`

```
SELECT hasnt_language( :language, :description );
SELECT hasnt_language( :language );
```

Parameters

`:language`

Name of the language.

`:description`

A short description of the test.

The inverse of `has_language()`, this function tests for the *absence* of a procedural language.

Table For One

Okay, you’re sure that your database has exactly the right schema and that all of the objects you need are there. So let’s take a closer look at tables. There are a lot of ways to look at tables, to make sure that they have all the columns, indexes, constraints, keys, and indexes they need. So we have the assertions to validate ’em.

`has_column()`

```
SELECT has_column( :schema, :table, :column, :description );
SELECT has_column( :table, :column, :description );
SELECT has_column( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of the column.

`:description`

A short description of the test.

Tests whether or not a column exists in a given table, view, materialized view or composite type. The first argument is the schema name, the second the table name, the third the column name, and the fourth is the test description. If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to "Column `:table.:column` should exist".

`hasnt_column()`

```
SELECT hasnt_column( :schema, :table, :column, :description );
SELECT hasnt_column( :table, :column, :description );
SELECT hasnt_column( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of the column.

`:description`

A short description of the test.

This function is the inverse of `has_column()`. The test passes if the specified column does *not* exist in the specified table, view, materialized view or composite type.

`col_not_null()`

```
SELECT col_not_null( :schema, :table, :column, :description );
SELECT col_not_null( :table, :column, :description );
SELECT col_not_null( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of the column.

`:description`

A short description of the test.

Tests whether the specified column has a `NOT NULL` constraint. The first argument is the schema name, the second the table name, the third the column name, and the fourth is the test description. If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to "Column `:table.:column` should be NOT NULL". Note that this test will fail with a useful diagnostic message if the table or column in question does not exist. But use `has_column()` to make sure the column exists first, eh?

col_is_null()

```
SELECT col_is_null( :schema, :table, :column, :description );
SELECT col_is_null( :table, :column, :description );
SELECT col_is_null( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of the column.

`:description`

A short description of the test.

This function is the inverse of `col_not_null()`: the test passes if the column does not have a `NOT NULL` constraint. The first argument is the schema name, the second the table name, the third the column name, and the fourth is the test description. If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to "Column `:table.:column` should allow NULL". Note that this test will fail with a useful diagnostic message if the table or column in question does not exist. But use `has_column()` to make sure the column exists first, eh?

col_has_default()

```
SELECT col_has_default( :schema, :table, :column, :description );
SELECT col_has_default( :table, :column, :description );
SELECT col_has_default( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of the column.

`:description`

A short description of the test.

Tests whether or not a column has a default value. Fails if the column doesn't have a default value. It will also fail if the column doesn't exist, and emit useful diagnostics to let you know:

```
# Failed test 136: "desc"
#     Column public.sometab.__asdfasdfs__ does not exist
```

`col_hasnt_default()`

```
SELECT col_hasnt_default( :schema, :table, :column, :description );
SELECT col_hasnt_default( :table, :column, :description );
SELECT col_hasnt_default( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of the column.

`:description`

A short description of the test.

This function is the inverse of `col_has_default()`. The test passes if the specified column does *not* have a default. It will still fail if the column does not exist, and emit useful diagnostics to let you know.

`col_type_is()`

```
SELECT col_type_is( :schema, :table, :column, :type_schema, :type, :d
SELECT col_type_is( :schema, :table, :column, :type_schema, :type );
```


```
SELECT col_type_is( :schema, :table, :column, :type, :description );
SELECT col_type_is( :schema, :table, :column, :type );
SELECT col_type_is( :table, :column, :type, :description );
SELECT col_type_is( :table, :column, :type );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of the column.

`:type_schema`

Schema in which to find the data type.

`:type`

Name of a data type.

`:description`

A short description of the test.

This function tests that the specified column is of a particular type. If it fails, it will emit diagnostics naming the actual type. The first argument is the schema name, the second the table name, the third the column name, the fourth the type's schema, the fifth the type, and the sixth is the test description.

If the table schema is omitted, the table must be visible in the search path. If the type schema is omitted, it must be visible in the search path; otherwise, the diagnostics will report the schema it's actually in. The schema can optinally be included in the `:type` argument, e.g., `'contrib.citext`.

If the test description is omitted, it will be set to "Column `:schema.:table.:column` should be type `:schema.:type`". Note that this test will fail if the table or column in question does not exist.

The type argument should be formatted as it would be displayed in the view of a table using the `\d` command in `psql`. For example, if you have a numeric column with a precision of 8, you should specify "numeric(8,0)". If you created a `varchar(64)` column, you should pass the type as "character varying(64)". Example:


```
SELECT col_type_is( 'myschema', 'sometable', 'somecolumn', 'numeric(1
```

If the test fails, it will output useful diagnostics. For example this test:

```
SELECT col_type_is( 'pg_catalog', 'pg_type', 'typname', 'text' );
```

Will produce something like this:

```
# Failed test 138: "Column pg_catalog.pg_type.typname should be type
#         have: name
#         want: text
```

It will even tell you if the test fails because a column doesn't exist or actually has no default. But use `has_column()` to make sure the column exists first, eh?

`col_default_is()`

```
SELECT col_default_is( :schema, :table, :column, :default, :descripti
SELECT col_default_is( :table, :column, :default, :description );
SELECT col_default_is( :table, :column, :default );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of the column.

`:default`

Default value expressed as a string.

`:description`

A short description of the test.

Tests the default value of a column. If it fails, it will emit diagnostics showing the actual default value. The first argument is the schema name, the second the table name, the third the column name, the fourth the default value, and the fifth is the test description. If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to "Column

`:table.:column` should default to `:default`". Note that this test will fail if the table or column in question does not exist.

The default argument must have an unambiguous type in order for the call to succeed. If you see an error such as 'ERROR: could not determine polymorphic type because input has type "unknown"', it's because you forgot to cast the expected value, probably a `NULL` (which, by the way, you can only properly test for in PostgreSQL 8.3 and later), to its proper type. IOW, this will fail:

```
SELECT col_default_is( 'tab', age, NULL );
```

But this will not:

```
SELECT col_default_is( 'tab', age, NULL::integer );
```

You can also test for functional defaults. Just specify the function call as a string:

```
SELECT col_default_is( 'user', 'created_at', 'now()' );
```

If the test fails, it will output useful diagnostics. For example, this test:

```
SELECT col_default_is(
    'pg_catalog',
    'pg_type',
    'typname',
    'foo',
    'check typname'
);
```

Will produce something like this:

```
# Failed test 152: "check typname"
#         have: NULL
#         want: foo
```

And if the test fails because the table or column in question does not exist, the diagnostics will tell you that, too. But you use `has_column()` and `col_has_default()` to test those conditions before you call `col_default_is()`, right? *Right???* Yeah, good, I thought so.

has_pk()

```
SELECT has_pk( :schema, :table, :description );
SELECT has_pk( :table, :description );
SELECT has_pk( :table );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:description`

A short description of the test.

Tests whether or not a table has a primary key. The first argument is the schema name, the second the table name, the the third is the test description. If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to "Table `:table` should have a primary key". Note that this test will fail if the table in question does not exist.

`hasnt_pk()`

```
SELECT hasnt_pk( :schema, :table, :description );
SELECT hasnt_pk( :table, :description );
SELECT hasnt_pk( :table );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:description`

A short description of the test.

This function is the inverse of `has_pk()`. The test passes if the specified primary key does *not* exist.

`has_fk()`

```
SELECT has_fk( :schema, :table, :description );
SELECT has_fk( :table, :description );
SELECT has_fk( :table );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:description`

A short description of the test.

Tests whether or not a table has a foreign key constraint. The first argument is the schema name, the second the table name, the the third is the test description. If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to "Table `:table` should have a foreign key constraint". Note that this test will fail if the table in question does not exist.

`hasnt_fk()`

```
SELECT hasnt_fk( :schema, :table, :description );
SELECT hasnt_fk( :table, :description );
SELECT hasnt_fk( :table );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table.

`:description`

A short description of the test.

This function is the inverse of `has_fk()`. The test passes if the specified foreign key does *not* exist.

`col_is_pk()`

```
SELECT col_is_pk( :schema, :table, :columns, :description );
SELECT col_is_pk( :schema, :table, :column, :description );
SELECT col_is_pk( :table, :columns, :description );
SELECT col_is_pk( :table, :column, :description );
SELECT col_is_pk( :table, :columns );
SELECT col_is_pk( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the primary key.

`:columns`

Array of the names of the primary key columns.

`:column`

Name of the primary key column.

`:description`

A short description of the test.

Tests whether the specified column or columns in a table is/are the primary key for that table. If it fails, it will emit diagnostics showing the actual primary key columns, if any. The first argument is the schema name, the second the table name, the third the column name or an array of column names, and the fourth is the test description. Examples:

```
SELECT col_is_pk( 'myschema', 'sometable', 'id' );
SELECT col_is_pk( 'persons',  ARRAY['given_name', 'surname'] );
```

If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to "Column `:table(:column)` should be a primary key". Note that this test will fail if the table or column in question does not exist.

If the test fails, it will output useful diagnostics. For example this test:

```
SELECT col_is_pk( 'pg_type', 'id' );
```

Will produce something like this:

```
# Failed test 178: "Column pg_type.id should be a primary key"
#         have: {}
#         want: {id}
```

col_isnt_pk()

```
SELECT col_isnt_pk( :schema, :table, :columns, :description );
SELECT col_isnt_pk( :schema, :table, :column, :description );
SELECT col_isnt_pk( :table, :columns, :description );
SELECT col_isnt_pk( :table, :column, :description );
SELECT col_isnt_pk( :table, :columns );
SELECT col_isnt_pk( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table not containing the primary key.

`:columns`

Array of the names of the primary key columns.

`:column`

Name of the primary key column.

`:description`

A short description of the test.

This function is the inverse of `col_is_pk()`. The test passes if the specified column or columns are not a primary key.

col_is_fk()

```
SELECT col_is_fk( :schema, :table, :columns, :description );
SELECT col_is_fk( :schema, :table, :column, :description );
SELECT col_is_fk( :table, :columns, :description );
SELECT col_is_fk( :table, :column, :description );
SELECT col_is_fk( :table, :columns );
SELECT col_is_fk( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the foreign key constraint.

`:columns`

Array of the names of the foreign key columns.

`:column`

Name of the foreign key column.

`:description`

A short description of the test.

Just like `col_is_fk()`, except that it test that the column or array of columns are a primary key. The diagnostics on failure are a bit different, too. Since the table might have more than one foreign key, the diagnostics simply list all of the foreign key constraint columns, like so:


```
#    Table widget has foreign key constraints on these columns:
#        {thingy_id}
#        {surname,given_name}
```

col_isnt_fk()

```
SELECT col_isnt_fk( :schema, :table, :columns, :description );
SELECT col_isnt_fk( :schema, :table, :column, :description );
SELECT col_isnt_fk( :table, :columns, :description );
SELECT col_isnt_fk( :table, :column, :description );
SELECT col_isnt_fk( :table, :columns );
SELECT col_isnt_fk( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table not containing the foreign key constraint.

`:columns`

Array of the names of the foreign key columns.

`:column`

Name of the foreign key column.

`:description`

A short description of the test.

This function is the inverse of `col_is_fk()`. The test passes if the specified column or columns are not a foreign key.

fk_ok()

```
SELECT fk_ok( :fk_schema, :fk_table,   :fk_columns, :pk_schema,  :pk_
SELECT fk_ok( :fk_schema, :fk_table,   :fk_columns, :pk_schema,  :pk_
SELECT fk_ok( :fk_table,  :fk_columns, :pk_table,   :pk_columns, :des
SELECT fk_ok( :fk_table,  :fk_columns, :pk_table,   :pk_columns );
SELECT fk_ok( :fk_schema, :fk_table,   :fk_column,  :pk_schema,  :pk_
SELECT fk_ok( :fk_schema, :fk_table,   :fk_column,  :pk_schema,  :pk_
SELECT fk_ok( :fk_table,  :fk_column,  :pk_table,   :pk_column,  :des
SELECT fk_ok( :fk_table,  :fk_column,  :pk_table,   :pk_column );
```

Parameters

`:fk_schema`

Schema in which to find the table with the foreign key

`:fk_table`

Name of a table containing the foreign key.

`:fk_columns`

Array of the names of the foreign key columns.

`:fk_column`

Name of the foreign key column.

`:pk_schema`

Schema in which to find the table with the primary key

`:pk_table`

Name of a table containing the primary key.

`:pk_columns`

Array of the names of the primary key columns.

`:pk_column`

Name of the primary key column.

`:description`

A short description of the test.

This function combines `col_is_fk()` and `col_is_pk()` into a single test that also happens to determine that there is in fact a foreign key relationship between the foreign and primary key tables. To properly test your relationships, this should be your main test function of choice.

The first three arguments are the schema, table, and column or array of columns that constitute the foreign key constraint. The schema name is optional, and the columns can be specified as a string for a single column or an array of strings for multiple columns. The next three arguments are the schema, table, and column or columns that constitute the corresponding primary key. Again, the schema is optional and the columns may be a string or array of strings (though of course it should have the same number of elements as the foreign key column argument). The seventh argument is an optional description If it's not included, it will be set to `:fk_schema.:fk_table(:fk_column)` should reference `:pk_column.pk_table(:pk_column)`. Some examples:

```
SELECT fk_ok( 'myschema', 'sometable', 'big_id', 'myschema', 'bigtabl
SELECT fk_ok(
    'contacts',
    ARRAY['person_given_name', 'person_surname'],
    'persons',
```


```
    ARRAY['given_name', 'surname'],
);
```

If the test fails, it will output useful diagnostics. For example this test:

```
SELECT fk_ok( 'contacts', 'person_id', 'persons', 'id' );
```

Will produce something like this:

```
# Failed test 178: "Column contacts(person_id) should reference perso
#         have: contacts(person_id) REFERENCES persons(id)"
#         want: contacts(person_nick) REFERENCES persons(nick)"
```

has_unique()

```
SELECT has_unique( :schema, :table, :description );
SELECT has_unique( :table, :description );
SELECT has_unique( :table );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the unique constraint.

`:description`

A short description of the test.

Tests whether or not a table has a unique constraint. The first argument is the schema name, the second the table name, the the third is the test description. If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to “Table `:table` should have a unique constraint”. Note that this test will fail if the table in question does not exist.

col_is_unique()

```
SELECT col_is_unique( schema, table, columns, description );
SELECT col_is_unique( schema, table, column, description );
SELECT col_is_unique( schema, table, columns );
SELECT col_is_unique( schema, table, column );
SELECT col_is_unique( table, columns, description );
SELECT col_is_unique( table, column, description );
SELECT col_is_unique( table, columns );
SELECT col_is_unique( table, column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the unique constraint.

`:columns`

Array of the names of the unique columns.

`:column`

Name of the unique column.

`:description`

A short description of the test.

Just like `col_is_pk()`, except that it test that the column or array of columns have a unique constraint on them. Examples:

```
SELECT col_is_unique( 'contacts', ARRAY['given_name', 'surname'] );
SELECT col_is_unique(
    'myschema', 'sometable', 'other_id',
    'myschema.sometable.other_id should be unique'
);
```

If you omit the description for the 3-argument version, you'll need to cast the table and column parameters to the `NAME` data type so that PostgreSQL doesn't resolve the function name as a description. For example:

```
SELECT col_is_unique( 'myschema', 'sometable'::name, 'other_id'::name
```

In the event of failure, the diagnostics will list the unique constraints that were actually found, if any:

```
Failed test 40: "users.email should be unique"
        have: {username}
              {first_name,last_name}
        want: {email}
```

has_check()

```
SELECT has_check( :schema, :table, :description );
SELECT has_check( :table, :description );
SELECT has_check( :table );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the check constraint.

`:description`

A short description of the test.

Tests whether or not a table has a check constraint. The first argument is the schema name, the second the table name, the the third is the test description. If the schema is omitted, the table must be visible in the search path. If the test description is omitted, it will be set to "Table `:table` should have a check constraint". Note that this test will fail if the table in question does not exist.

In the event of failure, the diagnostics will list the columns on the table that do have check constraints, if any:

```
Failed test 41: "users.email should have a check constraint"
        have: {username}
        want: {email}
```

col_has_check()

```
SELECT col_has_check( :schema, :table, :columns, :description );
SELECT col_has_check( :schema, :table, :column, :description );
SELECT col_has_check( :table, :columns, :description );
SELECT col_has_check( :table, :column, :description );
SELECT col_has_check( :table, :columns );
SELECT col_has_check( :table, :column );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the check constraint.

`:columns`

Array of the names of the check constraint columns.

`:column`

Name of the check constraint column.

`:description`

A short description of the test.

Just like `col_is_pk()`, except that it test that the column or array of columns have a check constraint on them.

index_is_unique()

```
SELECT index_is_unique( :schema, :table, :index, :description );
SELECT index_is_unique( :schema, :table, :index );
SELECT index_is_unique( :table, :index );
SELECT index_is_unique( :index );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the index.

`:index`

Name of the index.

`:description`

A short description of the test.

Tests whether an index is unique.

index_is_primary()

```
SELECT index_is_primary( :schema, :table, :index, :description );
SELECT index_is_primary( :schema, :table, :index );
SELECT index_is_primary( :table, :index );
SELECT index_is_primary( :index );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the index.

`:index`

Name of the index.

`:description`

A short description of the test.

Tests whether an index is on a primary key.

is_clustered()


```
SELECT is_clustered( :schema, :table, :index, :description );
SELECT is_clustered( :schema, :table, :index );
SELECT is_clustered( :table, :index );
SELECT is_clustered( :index );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the index.

`:index`

Name of the index.

`:description`

A short description of the test.

Tests whether a table is clustered on the given index. A table is clustered on an index when the SQL command `CLUSTER TABLE INDEXNAME` has been executed. Clustering reorganizes the table tuples so that they are stored on disk in the order defined by the index.

index_is_type()

```
SELECT index_is_type( :schema, :table, :index, :type, :description );
SELECT index_is_type( :schema, :table, :index, :type );
SELECT index_is_type( :table, :index, :type );
SELECT index_is_type( :index, :type );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Name of a table containing the index.

`:index`

Name of the index.

`:type`

The index Type.

`:description`

A short description of the test.

Tests to ensure that an index is of a particular type. At the time of this writing, the supported types are:

- btree
- hash
- gist
- gin

If the test fails, it will emit a diagnostic message with the actual index type, like so:

```
# Failed test 175: "Index idx_bar should be a hash index"
#         have: btree
#         want: hash
```

Feeling Funky

Perhaps more important than testing the database schema is testing your custom functions. Especially if you write functions that provide the interface for clients to interact with the database, making sure that they work will save you time in the long run. So check out these assertions to maintain your sanity.

can()

```
SELECT can( :schema, :functions, :description );
SELECT can( :schema, :functions );
SELECT can( :functions, :description );
SELECT can( :functions );
```

Parameters

`:schema`

Schema in which to find the functions.

`:functions`

Array of function names.

`:description`

A short description of the test.

Checks to be sure that `:schema` has `:functions` defined. This is subtly different from `functions_are()`. `functions_are()` fails if the functions defined in `:schema` are not exactly the functions defined in `:functions`. `can()`, on the other hand, just makes sure that `:functions` exist.

If `:schema` is omitted, then `can()` will look for functions defined in schemas defined in the search path. No matter how many functions are listed in `:functions`, a single call to `can()` counts as one test. If you want otherwise, call `can()` once for each function – or better yet, use `has_function()`. Example:

```
SELECT can( 'pg_catalog', ARRAY['upper', 'lower'] );
```

If any of the functions are not defined, the test will fail and the diagnostics will output a list of the functions that are missing, like so:

```
# Failed test 52: "Schema pg_catalog can"
#     pg_catalog.foo() missing
#     pg_catalog.bar() missing
```

function_lang_is()

```
SELECT function_lang_is( :schema, :function, :args, :language, :descr
SELECT function_lang_is( :schema, :function, :args, :language );
SELECT function_lang_is( :schema, :function, :language, :description
SELECT function_lang_is( :schema, :function, :language );
SELECT function_lang_is( :function, :args, :language, :description );
SELECT function_lang_is( :function, :args, :language );
SELECT function_lang_is( :function, :language, :description );
SELECT function_lang_is( :function, :language );
```

Parameters

`:schema`

Schema in which to find the function.

`:function`

Function name.

`:args`

Array of data types for the function arguments.

`:language`

Name of the procedural language.

`:description`

A short description of the test.

Tests that a particular function is implemented in a particular procedural language. The function name is required. If the `:schema` argument is omitted, then the function must be visible in the search path. If the `:args[]` argument is passed, then the function with that argument signature will be the one tested;

otherwise, a function with any signature will be checked (pass an empty array to specify a function with an empty signature). If the `:description` is omitted, a reasonable substitute will be created. Examples:

```
SELECT function_lang_is( 'myschema', 'foo',  ARRAY['integer', 'text']
SELECT function_lang_is( 'do_something', 'sql' );
SELECT function_lang_is( 'do_something', ARRAY['integer'], 'plpgsql'
SELECT function_lang_is( 'do_something', ARRAY['numeric'], 'plpgsql'
```

In the event of a failure, you'll useful diagnostics will tell you what went wrong, for example:

```
# Failed test 211: "Function mychema.eat(integer, text) should be wri
#         have: plpgsql
#         want: perl
```

If the function does not exist, you'll be told that, too.

```
# Failed test 212: "Function myschema.grab() should be written in sql
#     Function myschema.grab() does not exist
```

But then you check with `has_function()` first, right?

`function_returns()`

```
SELECT function_returns( :schema, :function, :args, :type, :descripti
SELECT function_returns( :schema, :function, :args, :type );
SELECT function_returns( :schema, :function, :type, :description );
SELECT function_returns( :schema, :function, :type );
SELECT function_returns( :function, :args, :type, :description );
SELECT function_returns( :function, :args, :type );
SELECT function_returns( :function, :type, :description );
SELECT function_returns( :function, :type );
```

Parameters

`:schema`

Schema in which to find the function.

`:function`

Function name.

`:args`

Array of data types for the function arguments.

`:Type`

Return value data type.

`:description`

A short description of the test.

Tests that a particular function returns a particular data type. The `:args[]` and `:type` arguments should be formatted as they would be displayed in the view of a function using the `\df` command in `psql`. For example, use “character varying” rather than “varchar”, and “boolean” rather than “bool”. For set returning functions, the `:type` argument should start with “setof ” (yes, lowercase). Examples:

```
SELECT function_returns( 'myschema', 'foo', ARRAY['integer', 'text'],
SELECT function_returns( 'do_something', 'setof bool' );
SELECT function_returns( 'do_something', ARRAY['integer'], 'boolean'
SELECT function_returns( 'do_something', ARRAY['numeric'], 'numeric'
```

If the `:schema` argument is omitted, then the function must be visible in the search path. If the `:args[]` argument is passed, then the function with that argument signature will be the one tested; otherwise, a function with any signature will be checked (pass an empty array to specify a function with an empty signature). If the `:description` is omitted, a reasonable substitute will be created.

In the event of a failure, you’ll useful diagnostics will tell you what went wrong, for example:

```
# Failed test 283: "Function oww(integer, text) should return integer
#         have: bool
#         want: integer
```

If the function does not exist, you’ll be told that, too.

```
# Failed test 284: "Function oui(integer, text) should return integer
#     Function oui(integer, text) does not exist
```

But then you check with `has_function()` first, right?

is_definer()


```
SELECT is_definer( :schema, :function, :args, :description );
SELECT is_definer( :schema, :function, :args );
SELECT is_definer( :schema, :function, :description );
SELECT is_definer( :schema, :function );
SELECT is_definer( :function, :args, :description );
SELECT is_definer( :function, :args );
SELECT is_definer( :function, :description );
SELECT is_definer( :function );
```

Parameters

`:schema`

Schema in which to find the function.

`:function`

Function name.

`:args`

Array of data types for the function arguments.

`:description`

A short description of the test.

Tests that a function is a security definer (i.e., a "setuid" function). If the `:schema` argument is omitted, then the function must be visible in the search path. If the `:args` argument is passed, then the function with that argument signature will be the one tested; otherwise, a function with any signature will be checked (pass an empty array to specify a function with an empty signature). If the `:description` is omitted, a reasonable substitute will be created. Examples:

```
SELECT is_definer( 'myschema', 'foo', ARRAY['integer', 'text'] );
SELECT is_definer( 'do_something' );
SELECT is_definer( 'do_something', ARRAY['integer'] );
SELECT is_definer( 'do_something', ARRAY['numeric'] );
```

If the function does not exist, a handy diagnostic message will let you know:

```
# Failed test 290: "Function nasty() should be security definer"
#     Function nasty() does not exist
```

But then you check with `has_function()` first, right?

is_strict()

```
SELECT is_strict( :schema, :function, :args, :description );
SELECT is_strict( :schema, :function, :args );
SELECT is_strict( :schema, :function, :description );
```


```
SELECT is_strict( :schema, :function );
SELECT is_strict( :function, :args, :description );
SELECT is_strict( :function, :args );
SELECT is_strict( :function, :description );
SELECT is_strict( :function );
```

Parameters

`:schema`

Schema in which to find the function.

`:function`

Function name.

`:args`

Array of data types for the function arguments.

`:description`

A short description of the test.

Tests that a function is a strict, meaning that the function returns null if any argument is null. If the `:schema` argument is omitted, then the function must be visible in the search path. If the `:args` argument is passed, then the function with that argument signature will be the one tested; otherwise, a function with any signature will be checked (pass an empty array to specify a function with an empty signature). If the `:description` is omitted, a reasonable substitute will be created. Examples:

```
SELECT is_strict( 'myschema', 'foo', ARRAY['integer', 'text'] );
SELECT is_strict( 'do_something' );
SELECT is_strict( 'do_something', ARRAY['integer'] );
SELECT is_strict( 'do_something', ARRAY['numeric'] );
```

If the function does not exist, a handy diagnostic message will let you know:

```
# Failed test 290: "Function nasty() should be strict"
#     Function nasty() does not exist
```

But then you check with `has_function()` first, right?

isnt_strict()

```
SELECT isnt_strict( :schema, :function, :args, :description );
SELECT isnt_strict( :schema, :function, :args );
SELECT isnt_strict( :schema, :function, :description );
SELECT isnt_strict( :schema, :function );
SELECT isnt_strict( :function, :args, :description );
```


```
SELECT isnt_strict( :function, :args );
SELECT isnt_strict( :function, :description );
SELECT isnt_strict( :function );
```

Parameters

`:schema`

Schema in which to find the function.

`:function`

Function name.

`:args`

Array of data types for the function arguments.

`:description`

A short description of the test.

This function is the inverse of `is_strict()`. The test passes if the specified function is not strict.

If the function does not exist, a handy diagnostic message will let you know:

```
# Failed test 290: "Function nasty() should be strict"
#     Function nasty() does not exist
```

But then you check with `has_function()` first, right?

is_aggregate()

```
SELECT is_aggregate( :schema, :function, :args, :description );
SELECT is_aggregate( :schema, :function, :args );
SELECT is_aggregate( :schema, :function, :description );
SELECT is_aggregate( :schema, :function );
SELECT is_aggregate( :function, :args, :description );
SELECT is_aggregate( :function, :args );
SELECT is_aggregate( :function, :description );
SELECT is_aggregate( :function );
```

Parameters

`:schema`

Schema in which to find the function.

`:function`

Function name.

`:args`

Array of data types for the function arguments.

`:description`

A short description of the test.

Tests that a function is an aggregate function. If the `:schema` argument is omitted, then the function must be visible in the search path. If the `:args[]` argument is passed, then the function with that argument signature will be the one tested; otherwise, a function with any signature will be checked (pass an empty array to specify a function with an empty signature). If the `:description` is omitted, a reasonable substitute will be created. Examples:

```
SELECT is_aggregate( 'myschema', 'foo',  ARRAY['integer', 'text'] );
SELECT is_aggregate( 'do_something' );
SELECT is_aggregate( 'do_something', ARRAY['integer'] );
SELECT is_aggregate( 'do_something', ARRAY['numeric'] );
```

If the function does not exist, a handy diagnostic message will let you know:

```
# Failed test 290: "Function nasty() should be strict"
#     Function nasty() does not exist
```

But then you check with `has_function()` first, right?

volatility_is()

```
SELECT volatility_is( :schema, :function, :args, :volatility, :descri
SELECT volatility_is( :schema, :function, :args, :volatility );
SELECT volatility_is( :schema, :function, :volatility, :description );
SELECT volatility_is( :schema, :function, :volatility );
SELECT volatility_is( :function, :args, :volatility, :description );
SELECT volatility_is( :function, :args, :volatility );
SELECT volatility_is( :function, :volatility, :description );
SELECT volatility_is( :function, :volatility );
```

Parameters

`:schema`

Schema in which to find the function.

`:function`

Function name.

`:args`

Array of data types for the function arguments.

`:volatility`

Volatility level.

`:description`

A short description of the test.

Tests the volatility of a function. Supported volatilities are “volatile”, “stable”, and “immutable”. Consult the `CREATE FUNCTION` documentation for details. The function name is required. If the `:schema` argument is omitted, then the function must be visible in the search path. If the `:args[]` argument is passed, then the function with that argument signature will be the one tested; otherwise, a function with any signature will be checked (pass an empty array to specify a function with an empty signature). If the `:description` is omitted, a reasonable substitute will be created. Examples:

```
SELECT volatility_is( 'myschema', 'foo',  ARRAY['integer', 'text'], '
SELECT volatility_is( 'do_something', 'immutable' );
SELECT volatility_is( 'do_something', ARRAY['integer'], 'stable' );
SELECT volatility_is( 'do_something', ARRAY['numeric'], 'volatile' );
```

In the event of a failure, you’ll useful diagnostics will tell you what went wrong, for example:

```
# Failed test 211: "Function mychema.eat(integer, text) should be IMM
#         have: VOLATILE
#         want: IMMUTABLE
```

If the function does not exist, you’ll be told that, too.

```
# Failed test 212: "Function myschema.grab() should be IMMUTABLE"
#     Function myschema.grab() does not exist
```

But then you check with `has_function()` first, right?

`trigger_is()`

```
SELECT trigger_is( :schema, :table, :trigger, :func_schema, :function
SELECT trigger_is( :schema, :table, :trigger, :func_schema, :function
SELECT trigger_is( :table, :trigger, :function, :description );
SELECT trigger_is( :table, :trigger, :function );
```

Parameters

`:schema`

Schema in which to find the table.

`:table`

Table in which to find the trigger.

`:trigger`

Trigger name.

`:func_schema`

Schema in which to find the trigger function.

`:function`

Function name.

`:description`

A short description of the test.

Tests that the specified trigger calls the named function. If not, it outputs a useful diagnostic:

```
# Failed test 31: "Trigger set_users_pass should call hash_password()
#         have: hash_pass
#         want: hash_password
```

Database Deets

Tables and functions aren't the only objects in the database, as you well know. These assertions close the gap by letting you test the attributes of other database objects.

`language_is_trusted()`

```
SELECT language_is_trusted( language, description );
SELECT language_is_trusted( language );
```

Parameters

`:language`

Name of a procedural language.

`:description`

A short description of the test.

Tests that the specified procedural language is trusted. See the CREATE LANGUAGE documentation for details on trusted and untrusted procedural languages. If the `:description` argument is not passed, a suitably useful default will be created.

In the event that the language in question does not exist in the database, `language_is_trusted()` will emit a diagnostic message to alert you to this fact, like so:

```
# Failed test 523: "Procedural language plomgwtf should be trusted"
#     Procedural language plomgwtf does not exist
```

But you really ought to call `has_language()` first so that you never get that far.

enum_has_labels()

```
SELECT enum_has_labels( :schema, :enum, :labels, :description );
SELECT enum_has_labels( :schema, :enum, :labels );
SELECT enum_has_labels( :enum, :labels, :description );
SELECT enum_has_labels( :enum, :labels );
```

Parameters

`:schema`

Schema in which to find the enum.

`:enum`

Enum name.

`:labels`

An array of the enum labels.

`:description`

A short description of the test.

This function tests that an enum consists of an expected list of labels. Enums are supported in PostgreSQL 8.3 or higher. The first argument is a schema name, the second an enum name, the third an array of enum labels, and the fourth a description. Example:

```
SELECT enum_has_labels( 'myschema', 'someenum', ARRAY['foo', 'bar'] )
```

If you omit the schema, the enum must be visible in the search path. If you omit the test description, it will be set to “Enum `:enum` should have labels (`:labels`)”.

domain_type_is()

```
SELECT domain_type_is( :schema, :domain, :type_schema, :type, :descri
SELECT domain_type_is( :schema, :domain, :type_schema, :type );
SELECT domain_type_is( :schema, :domain, :type, :description );
SELECT domain_type_is( :schema, :domain, :type );
```


```
SELECT domain_type_is( :domain, :type, :description );
SELECT domain_type_is( :domain, :type );
```

Parameters

`:schema`

Schema in which to find the domain.

`:domain`

Domain name.

`:type_schema`

Schema in which to find the data type.

`:type`

Domain data type.

`:description`

A short description of the test.

Tests the data type underlying a domain. The first two arguments are the schema and name of the domain. The second two are the schema and name of the type that the domain should extend. The fifth argument is a description. If there is no description, a reasonable default description will be created.

The schema arguments are also optional. However, if there is no `:schema` argument, there cannot be a `:type_schema` argument, either, though the schema can be included in the `type` argument, e.g., `contrib.citext`.

For the 3- and 4-argument forms with schemas, cast the schemas to `NAME` to avoid ambiguities. Example:

```
SELECT domain_type_is(
    'public'::name, 'us_postal_code',
    'public'::name, 'text'
);
```

If the data type does not match the type that the domain extends, the test will fail and output diagnostics like so:

```
# Failed test 631: "Domain public.us_postal_code should extend type p
#         have: public.text
#         want: public.integer
```

If the domain in question does not actually exist, the test will fail with diagnostics that tell you so:

```
# Failed test 632: "Domain public.zip_code should extend type public.
#    Domain public.zip_code does not exist
```

domain_type_isnt()

```
SELECT domain_type_isnt( :schema, :domain, :type_schema, :type, :desc
SELECT domain_type_isnt( :schema, :domain, :type_schema, :type );
SELECT domain_type_isnt( :schema, :domain, :type, :description );
SELECT domain_type_isnt( :schema, :domain, :type );
SELECT domain_type_isnt( :domain, :type, :description );
SELECT domain_type_isnt( :domain, :type );
```

Parameters

`:schema`

Schema in which to find the domain.

`:domain`

Domain name.

`:type_schema`

Schema in which to find the data type.

`:type`

Domain data type.

`:description`

A short description of the test.

The inverse of `domain_type_is()`, this function tests that a domain does *not* extend a particular data type. For example, a US postal code domain should probably extned the `text` type, not `integer`, since leading 0s are valid and required. Example:

```
SELECT domain_type_isnt(
    'public', 'us_postal_code',
    'public', 'integer',
    'The us_postal_code domain should not extend the integer type'
);
```

The arguments are the same as for `domain_type_is()`.

cast_context_is()


```
SELECT cast_context_is( :source_type, :target_type, :context, :descri
SELECT cast_context_is( :source_type, :target_type, :context );
```

Parameters

`:source_type`

The type cast from.

`:target_type`

The type cast to.

`:context`

The context for the cast, one of “implicit”, “assignment”, or “explicit”.

Test that a cast from a source to a target data type has a particular context. Example:

```
SELECT cast_context_is( 'integer', 'bigint', 'implicit' );
```

The data types should be passed as they are displayed by `pg_catalog.format_type()`. For example, you would need to pass “character varying”, and not “VARCHAR”.

The The supported contexts are “implicit”, “assignment”, and “explicit”. You can also just pass in “i”, “a”, or “e”. Consult the PostgreSQL `CREATE CAST` documentation for the differences between these contexts (hint: they correspond to the default context, `AS IMPLICIT`, and `AS ASSIGNMENT`). If you don’t supply a test description, pgTAP will create a reasonable one for you.

Test failure will result in useful diagnostics, such as:

```
# Failed test 124: "Cast ("integer" AS "bigint") context should be ex
#         have: implicit
#         want: explicit
```

If the cast doesn’t exist, you’ll be told that, too:

```
# Failed test 199: "Cast ("integer" AS foo) context should be explici
#     Cast ("integer" AS foo) does not exist
```

But you’ve already used `has_cast()` to make sure of that, right?

`is_superuser()`


```
SELECT is_superuser( :user, :description );
SELECT is_superuser( :user );
```

Parameters

`:user`

Name of a PostgreSQL user.

`:description`

A short description of the test.

Tests that a database user is a super user. If the description is omitted, it will default to "User `:user` should be a super user". Example:

```
SELECT is_superuser('theory' ;
```

If the user does not exist in the database, the diagnostics will say so.

`isnt_superuser()`

```
SELECT is_superuser(
    'dr_evil',
    'User "dr_evil" should not be a super user'
);
```

Parameters

`:user`

Name of a PostgreSQL user.

`:description`

A short description of the test.

The inverse of `is_superuser()`, this function tests that a database user is *not* a super user. Note that if the named user does not exist in the database, the test is still considered a failure, and the diagnostics will say so.

`is_member_of()`

```
SELECT is_member_of( :role, :members, :description );
SELECT is_member_of( :role, :members );
SELECT is_member_of( :role, :member, :description );
SELECT is_member_of( :role, :member );
```

Parameters

`:role`

Name of a PostgreSQL group role.

`:members`

Array of names of roles that should be members of the group role.

`:member`

Name of a role that should be a member of the group role.

`:description`

A short description of the test.

SELECT is_member_of(‘sweeties’, ‘anna’ ‘Anna should be a sweetie’);

SELECT is_member_of(‘meanies’, ARRAY[‘dr_evil’, ‘dr_no’]);

Checks whether a group role contains a member role or all of an array of member roles. If the description is omitted, it will default to “Should have members of role `:role`.” On failure, `is_member_of()` will output diagnostics listing the missing member roles, like so:

```
# Failed test 370: "Should have members of role meanies"
#     Members missing from the meanies role:
#         theory
#         agliodbs
```

If the group role does not exist, the diagnostics will tell you that, instead. But you use `has_role()` to make sure the role exists before you check its members, don’t you? Of course you do.

rule_is_instead()

```
SELECT rule_is_instead( :schema, :table, :rule, :description );
SELECT rule_is_instead( :schema, :table, :rule );
SELECT rule_is_instead( :table, :rule, :description );
SELECT rule_is_instead( :table, :rule );
```

Parameters

`:schema`

Name of a schema in which to find the table.

`:table`

Name of the table to which the rule is applied.

`:rule`

A rule name.

`:description`

A short description of the test.

Checks whether a rule on the specified relation is an `INSTEAD` rule. See the `CREATE RULE` Documentation for details. If the `:schema` argument is omitted, the relation must be visible in the search path. If the `:description` argument is omitted, an appropriate description will be created. An example:

```
SELECT rule_is_instead('public', 'users', 'on_insert');
```

In the event that the test fails because the rule in question does not actually exist, you will see an appropriate diagnostic such as:

```
# Failed test 625: "Rule on_insert on relation public.users should be
#     Rule on_insert does not exist
```

`rule_is_on()`

```
SELECT rule_is_on( :schema, :table, :rule, :event, :description );
SELECT rule_is_on( :schema, :table, :rule, :event );
SELECT rule_is_on( :table, :rule, :event, :description );
SELECT rule_is_on( :table, :rule, :event );
```

Parameters

`:schema`

Name of a schema in which to find the table.

`:table`

Name of the table to which the rule is applied.

`:rule`

A rule name.

`:event`

Name of a rule event, one of “SELECT”, “INSERT”, “UPDATE”, or “DELETE”.

`:description`

A short description of the test.

Tests the event for a rule, which may be one of “SELECT”, “INSERT”, “UPDATE”, or “DELETE”. For the `:event` argument, you can specify the name of the event in any case, or even with a single letter (“s”, “i”, “u”, or “d”). If the `:schema` argument is omitted, then the table must be visible in the search path. If the `:description` is omitted, a reasonable default will be created. Example:


```
SELECT rule_is_on('public', 'users', 'on_insert', 'INSERT');
```

If the test fails, you'll see useful diagnostics, such as:

```
# Failed test 133: "Rule ins_me should be on INSERT to public.widgets
#         have: UPDATE
#         want: INSERT
```

If the rule in question does not exist, you'll be told that, too:

```
# Failed test 134: "Rule upd_me should be on UPDATE to public.widgets
#     Rule upd_me does not exist on public.widgets
```

But then you run `has_rule()` first, don't you?

Who owns me?

After testing the availability of several objects, we often need to know who owns an object.

db_owner_is ()

```
SELECT db_owner_is ( :dbname, :user, :description );
SELECT db_owner_is ( :dbname, :user );
```

Parameters

`:dbname`

Name of a database.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of the database. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT db_owner_is( 'mydb', 'someuser', 'mydb should be owned by some
SELECT db_owner_is( current_database(), current_user );
```

In the event that the test fails because the database in question does not actually exist, you will see an appropriate diagnostic such as:


```
# Failed test 16: "Database foo should be owned by www"
#     Database foo does not exist
```

If the test fails because the database is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Database bar should be owned by root"
#         have: postgres
#         want: root
```

schema_owner_is ()

```
SELECT schema_owner_is ( :schemaname, :user, :description );
SELECT schema_owner_is ( :schemaname, :user );
```

Parameters

`:schemaname`

Name of a schema.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of the schema. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT schema_owner_is( 'myschema', 'someuser', 'myschema should be o
SELECT schema_owner_is( current_schema(), current_user );
```

In the event that the test fails because the schema in question does not actually exist, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Schema foo should be owned by www"
#     Schema foo does not exist
```

If the test fails because the schema is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Schema bar should be owned by root"
#         have: postgres
#         want: root
```

tablespace_owner_is ()


```
SELECT tablespace_owner_is ( :tablespacename, :user, :description );
SELECT tablespace_owner_is ( :tablespacename, :user );
```

Parameters

`:tablespacename`

Name of a tablespace.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of the tablespace. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT tablespace_owner_is( 'myts', 'joe', 'Joe has mytablespace' );
SELECT tablespace_owner_is( 'pg_default', current_user );
```

In the event that the test fails because the tablespace in question does not actually exist, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Tablespace ssd should be owned by www"
#     Tablespace ssd does not exist
```

If the test fails because the tablespace is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Tablespace raid_hds should be owned by root"
#         have: postgres
#         want: root
```

`relation_owner_is ()`

```
SELECT relation_owner_is ( :schema, :relation, :user, :description );
SELECT relation_owner_is ( :relation, :user, :description );
SELECT relation_owner_is ( :schema, :relation, :user );
SELECT relation_owner_is ( :relation, :user );
```

Parameters

`:schema`

Name of a schema in which find to the `:relation`.

`:relation`

Name of a relation.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a relation. Relations are tables, views, materialized views, seqences, composite types, foreign tables, and toast tables. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT relation_owner_is(
    'public', 'mytable', 'someuser',
    'mytable should be owned by someuser'
);
SELECT relation_owner_is( current_schema(), 'mysequence', current_use
```

In the event that the test fails because the relation in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Relation foo should be owned by www"
#     Relation foo does not exist
```

If the test fails because the relation is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Relation bar should be owned by root"
#         have: postgres
#         want: root
```

table_owner_is ()

```
SELECT table_owner_is ( :schema, :table, :user, :description );
SELECT table_owner_is ( :table, :user, :description );
SELECT table_owner_is ( :schema, :table, :user );
SELECT table_owner_is ( :table, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:table`.

`:table`

Name of a table.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a table. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT table_owner_is(
    'public', 'mytable', 'someuser',
    'mytable should be owned by someuser'
);
SELECT table_owner_is( 'widgets', current_user );
```

Note that this function will not recognize foreign tables; use `foreign_table_owner_is()` to test for the presence of foreign tables.

In the event that the test fails because the table in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Table foo should be owned by www"
#     Table foo does not exist
```

If the test fails because the table is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Table bar should be owned by root"
#         have: postgres
#         want: root
```

view_owner_is ()

```
SELECT view_owner_is ( :schema, :view, :user, :description );
SELECT view_owner_is ( :view, :user, :description );
SELECT view_owner_is ( :schema, :view, :user );
SELECT view_owner_is ( :view, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:view`.

`:view`

Name of a view.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a view. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT view_owner_is(
    'public', 'myview', 'someuser',
    'myview should be owned by someuser'
);
SELECT view_owner_is( 'widgets', current_user );
```

In the event that the test fails because the view in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "View foo should be owned by www"
#     View foo does not exist
```

If the test fails because the view is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "View bar should be owned by root"
#         have: postgres
#         want: root
```

materialized_view_owner_is ()

```
SELECT materialized_view_owner_is ( :schema, :materialized_view, :use
SELECT materialized_view_owner_is ( :materialized_view, :user, :descr
SELECT materialized_view_owner_is ( :schema, :materialized_view, :use
SELECT materialized_view_owner_is ( :materialized_view, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:materialized_view`.

`:materialized_view`

Name of a materialized view.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a materialized view. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT view_owner_is(
    'public', 'mymaterializedview', 'someuser',
    'mymaterializedview should be owned by someuser'
);
SELECT materialized_view_owner_is( 'widgets', current_user );
```

In the event that the test fails because the materialized view in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Materialized view foo should be owned by www"
#     Materialized view foo does not exist
```

If the test fails because the materialized view is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Materialized view bar should be owned by root"
#         have: postgres
#         want: root
```

sequence_owner_is ()

```
SELECT sequence_owner_is ( :schema, :sequence, :user, :description );
SELECT sequence_owner_is ( :sequence, :user, :description );
SELECT sequence_owner_is ( :schema, :sequence, :user );
SELECT sequence_owner_is ( :sequence, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:sequence`.

`:sequence`

Name of a sequence.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a sequence. If the `:description` argument is omitted, an appropriate description will be created. Examples:


```
SELECT sequence_owner_is(
    'public', 'mysequence', 'someuser',
    'mysequence should be owned by someuser'
);
SELECT sequence_owner_is( 'widgets', current_user );
```

In the event that the test fails because the sequence in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Sequence foo should be owned by www"
#     Sequence foo does not exist
```

If the test fails because the sequence is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Sequence bar should be owned by root"
#         have: postgres
#         want: root
```

composite_owner_is ()

```
SELECT composite_owner_is ( :schema, :composite, :user, :description
SELECT composite_owner_is ( :composite, :user, :description );
SELECT composite_owner_is ( :schema, :composite, :user );
SELECT composite_owner_is ( :composite, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:composite` type.

`:composite`

Name of a composite type.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a composite. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT composite_owner_is(
    'public', 'mycomposite', 'someuser',
    'mycomposite should be owned by someuser'
```


```
);
SELECT composite_owner_is( 'widgets', current_user );
```

In the event that the test fails because the composite in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Composite type foo should be owned by www"
#     Composite type foo does not exist
```

If the test fails because the composite is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Composite type bar should be owned by root"
#         have: postgres
#         want: root
```

foreign_table_owner_is ()

```
SELECT foreign_table_owner_is ( :schema, :foreign_table, :user, :desc
SELECT foreign_table_owner_is ( :foreign_table, :user, :description )
SELECT foreign_table_owner_is ( :schema, :foreign_table, :user );
SELECT foreign_table_owner_is ( :foreign_table, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:foreign_table`.

`:foreign_table`

Name of a foreign table.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a foreign table. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT foreign_table_owner_is(
    'public', 'mytable', 'someuser',
    'mytable should be owned by someuser'
);
SELECT foreign_table_owner_is( 'widgets', current_user );
```

In the event that the test fails because the foreign table in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Foreign table foo should be owned by www"
#     Foreign table foo does not exist
```

If the test fails because the foreign table is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Foreign table bar should be owned by root"
#         have: postgres
#         want: root
```

index_owner_is ()

```
SELECT index_owner_is ( :schema, :table, :index, :user, :description
SELECT index_owner_is ( :table, :index, :user, :description );
SELECT index_owner_is ( :schema, :table, :index, :user );
SELECT index_owner_is ( :table, :index, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:table, :index`.

`:table`

Name of a table.

`:table`

Name of an index on the table.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of an index. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT index_owner_is(
    'public', 'mytable', 'idx_name', 'someuser',
    'Index "idx_name" on mytable should be owned by someuser'
);
SELECT index_owner_is( 'widgets', 'widgets_pkey', current_user );
```

In the event that the test fails because the index in question does not actually exist, or the table or schema it's on does not exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Index idx_foo should be owned by root"
#     Index idx_foo on table darfoo not found
```

If the test fails because the table is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Index idx_foo on table bar should be owned by bob"
#         have: postgres
#         want: bob
```

function_owner_is ()

```
SELECT function_owner_is ( :schema, :function, :args, :user, :descrip
SELECT function_owner_is ( :function, :args, :user, :description );
SELECT function_owner_is ( :schema, :function, :args, :user );
SELECT function_owner_is ( :function, :args, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:function`.

`:function`

Name of a function.

`:args`

Array of data types of the function arguments.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a function. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT function_owner_is(
    'public', 'frobulate', ARRAY['integer', 'text'], 'someuser',
    'public.frobulate(integer, text) should be owned by someuser'
);
SELECT function_owner_is( 'masticate', ARRAY['text'], current_user );
```

In the event that the test fails because the function in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Function foo() should be owned by www"
#     Function foo() does not exist
```

If the test fails because the function is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "Function bar() should be owned by root"
#         have: postgres
#         want: root
```

language_owner_is ()

```
SELECT language_owner_is ( :languagename, :user, :description );
SELECT language_owner_is ( :languagename, :user );
```

Parameters

`:languagename`

Name of a language.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a procedural language. If the `:description` argument is omitted, an appropriate description will be created. Works on PostgreSQL 8.3 and higher. Examples:

```
SELECT language_owner_is( 'plpgsql', 'larry', 'Larry should own plpgs
SELECT language_owner_is( 'plperl', current_user );
```

In the event that the test fails because the language in question does not actually exist, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Language pllolcode should be owned by meow"
#     Language pllolcode does not exist
```

If the test fails because the language is not owned by the specified user, the diagnostics will look something like:


```
# Failed test 17: "Language plruby should be owned by mats"
#         have: postgres
#         want: mats
```

opclass_owner_is ()

```
SELECT opclass_owner_is ( :schema, :opclass, :user, :description );
SELECT opclass_owner_is ( :opclass, :user, :description );
SELECT opclass_owner_is ( :schema, :opclass, :user );
SELECT opclass_owner_is ( :opclass, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:opclass`.

`:opclass`

Name of an operator class.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of an operator class. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT opclass_owner_is(
    'pg_catalog', 'int4_ops', 'postgres',
    'Operator class int4_ops should be owned by postgres'
);
SELECT opclass_owner_is( 'my_ops', current_user );
```

In the event that the test fails because the operator class in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "operator class foo should be owned by www"
#     operator class foo does not exist
```

If the test fails because the operator class is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "operator class bar should be owned by root"
#         have: postgres
#         want: root
```

type_owner_is ()


```
SELECT type_owner_is ( :schema, :type, :user, :description );
SELECT type_owner_is ( :type, :user, :description );
SELECT type_owner_is ( :schema, :type, :user );
SELECT type_owner_is ( :type, :user );
```

Parameters

`:schema`

Name of a schema in which to find the `:type`.

`:type`

Name of a type.

`:user`

Name of a user.

`:description`

A short description of the test.

Tests the ownership of a data type. If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT type_owner_is(
    'pg_catalog', 'int4', 'postgres',
    'type int4 should be owned by postgres'
);
SELECT type_owner_is( 'us_postal_code', current_user );
```

In the event that the test fails because the type in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "type uk_postal_code should be owned by www"
#     type uk_postal_code does not exist
```

If the test fails because the type is not owned by the specified user, the diagnostics will look something like:

```
# Failed test 17: "type us_postal_code should be owned by root"
#         have: postgres
#         want: root
```

Prvileged Access

So we know who owns the objects. But what about other roles? Can they access database objects? Let's find out!

`database_privs_are()`


```
SELECT database_privs_are ( :db, :role, :privileges, :description );
SELECT database_privs_are ( :db, :role, :privileges );
```

Parameters

`:db`

Name of a database.

`:role`

Name of a user or group role.

`:privileges`

An array of table privileges the role should be granted to the database.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a database. The available database privileges are:

- CREATE
- CONNECT
- TEMPORARY

Although CONNECT is not available before PostgreSQL 8.2.

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT database_privs_are(
    'flipr', 'fred', ARRAY['CONNECT', 'TEMPORARY'],
    'Fred should be granted CONNECT and TERMPORARY on db "flipr"'
);
SELECT database_privs_are( 'dept_corrections', ARRAY['CREATE'] );
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted CREATE on database bank
#     Extra privileges:
#           CONNECT
#           TEMPORARY
```

Likewise if the role is not granted some of the specified permissions on the database:


```
# Failed test 15: "Role kurk should be granted CONNECT, TEMPORARY on
#     Missing privileges:
#           CREATE
```

In the event that the test fails because the database in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted CONNECT on database ma
#     Database maindb does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted CONNECT, CREATE on dat
#     Role slim does not exist
```

tablespace_privs_are()

```
SELECT tablespace_privs_are ( :tablespace, :role, :privileges, :descr
SELECT tablespace_privs_are ( :tablespace, :role, :privileges );
```

Parameters

`:tablespace`

Name of a tablespace.

`:role`

Name of a user or group role.

`:privileges`

An array of table privileges the role should be granted to the tablespace.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a tablespace. The available function privileges are:

- CREATE

If the `:description` argument is omitted, an appropriate description will be created. Examples:


```
SELECT tablespace_privs_are(
    'ssd', 'fred', ARRAY['CREATE'],
    'Fred should be granted CREATE on tablespace "ssd"'
);
SELECT tablespace_privs_are( 'san', ARRAY['CREATE'] );
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted no privileges on tables
#     Extra privileges:
#           CREATE
```

Likewise if the role is not granted some of the specified permissions on the tablespace:

```
# Failed test 15: "Role kurk should be granted USAGE on ssd"
#     Missing privileges:
#           CREATE
```

In the event that the test fails because the tablespace in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted CREATE on tablespace t
#     Tablespace tape does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted CREATE on san"
#     Role slim does not exist
```

schema_privs_are()

```
SELECT schema_privs_are ( :schema, :role, :privileges, :description );
SELECT schema_privs_are ( :schema, :role, :privileges );
```

Parameters

`:schema`

Name of a schema.

`:role`

Name of a user or group role.

`:privileges`

An array of table privileges the role should be granted to the schema.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a schema. The available schema privileges are:

- CREATE
- USAGE

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT schema_privs_are(
    'flipr', 'fred', ARRAY['CREATE', 'USAGE'],
    'Fred should be granted CREATE and USAGE on schema "flipr"'
);
SELECT schema_privs_are( 'hr', ARRAY['USAGE'] );
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted no privileges on schema
#     Extra privileges:
#           CREATE
#           USAGE
```

Likewise if the role is not granted some of the specified permissions on the schema:

```
# Failed test 15: "Role kurk should be granted CREATE, USAGE on schem
#     Missing privileges:
#           CREATE
```

In the event that the test fails because the schema in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted USAGE on schema main"
#     Schema main does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted CREATE, USAGE on schem
#     Role slim does not exist
```

table_privs_are()

```
SELECT table_privs_are ( :schema, :table, :role, :privileges, :descri
SELECT table_privs_are ( :schema, :table, :role, :privileges );
SELECT table_privs_are ( :table, :role, :privileges, :description );
SELECT table_privs_are ( :table, :role, :privileges );
```

Parameters

`:schema`

Name of a schema in which to find the table.

`:table`

Name of a table.

`:role`

Name of a user or group role.

`:privileges`

An array of table privileges the role should be granted to the table.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a table. The available table privileges are:

- DELETE
- INSERT
- REFERENCES
- RULE
- SELECT
- TRIGGER
- TRUNCATE
- UPDATE

Note that the privilege RULE is not available aftter PostgreSQL 8.1, and that TRIGGER was added in 8.4.

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT table_privs_are(
    'public', 'frobulate', 'fred', ARRAY['SELECT', 'DELETE'],
    'Fred should be able to select and delete on frobulate'
);
SELECT table_privs_are( 'widgets', 'slim', ARRAY['INSERT', 'UPDATE']
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted SELECT on widgets"
#     Extra privileges:
#           DELETE
#           INSERT
#           UPDATE
```

Likewise if the role is not granted some of the specified permissions on the table:

```
# Failed test 15: "Role kurk should be granted SELECT, INSERT, UPDATE
#     Missing privileges:
#           INSERT
#           UPDATE
```

In the event that the test fails because the table in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted SELECT on widgets"
#     Table widgets does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted SELECT on widgets"
#     Role slim does not exist
```

sequence_privs_are()

```
SELECT sequence_privs_are ( :schema, :sequence, :role, :privileges, :
SELECT sequence_privs_are ( :schema, :sequence, :role, :privileges );
SELECT sequence_privs_are ( :sequence, :role, :privileges, :descripti
SELECT sequence_privs_are ( :sequence, :role, :privileges );
```

Parameters

`:schema`

Name of a schema in which to find the sequence.

`:sequence`

Name of a sequence.

`:role`

Name of a user or group role.

`:privileges`

An array of sequence privileges the role should be granted to the sequence.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a sequence. The available sequence privileges are:

- SELECT
- UPDATE
- USAGE

Note that sequence privileges were added in PostgreSQL 9.0, so this function will likley throw an exception on earlier versions.

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT sequence_privs_are(
    'public', 'seq_ids', 'fred', ARRAY['SELECT', 'UPDATE'],
    'Fred should be able to select and update seq_ids'
);
SELECT sequence_privs_are( 'seq_u', 'slim', ARRAY['USAGE'] );
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted SELECT on seq_foo_id"
#     Extra privileges:
#           UPDATE
#           USAGE
```

Likewise if the role is not granted some of the specified permissions on the sequence:


```
# Failed test 15: "Role kurk should be granted USAGE on seq_widgets"
#     Missing privileges:
#           SELECT
#           UPDATE
```

In the event that the test fails because the sequence in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted SELECT on seq_widgets"
#     Sequence widgets does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted SELECT on seq_widgets"
#     Role slim does not exist
```

any_column_privs_are()

```
SELECT any_column_privs_are ( :schema, :table, :role, :privileges, :d
SELECT any_column_privs_are ( :schema, :table, :role, :privileges );
SELECT any_column_privs_are ( :table, :role, :privileges, :descriptio
SELECT any_column_privs_are ( :table, :role, :privileges );
```

Parameters

`:schema`

Name of a schema in which to find the table.

`:table`

Name of a table.

`:role`

Name of a user or group role.

`:privileges`

An array of table privileges the role should be granted to the table.

`:description`

A short description of the test.

Tests the privileges granted to access one or more of the columns in a table. The available column privileges are:

- INSERT

- REFERENCES
- SELECT
- UPDATE

Note that column privileges were added in PostgreSQL 8.4, so this function will likley throw an exception on earlier versions.

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT any_column_privs_are(
    'public', 'frobulate', 'fred', ARRAY['SELECT', 'UPDATE'],
    'Fred should be able to select and update columns in frobulate'
);
SELECT any_column_privs_are( 'widgets', 'slim', ARRAY['INSERT', 'UPDA
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted SELECT on columns in wi
#     Extra privileges:
#           INSERT
#           UPDATE
```

Likewise if the role is not granted some of the specified permissions on the table:

```
# Failed test 15: "Role kurk should be granted SELECT, INSERT, UPDATE
#     Missing privileges:
#           INSERT
#           UPDATE
```

In the event that the test fails because the table in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted SELECT on columns in w
#     Table widgets does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted SELECT on columns in w
#     Role slim does not exist
```

column_privs_are()

```
SELECT column_privs_are ( :schema, :table, :column, :role, :privilege
SELECT column_privs_are ( :schema, :table, :column, :role, :privilege
SELECT column_privs_are ( :table, :column, :role, :privileges, :descr
SELECT column_privs_are ( :table, :column, :role, :privileges );
```

Parameters

`:schema`

Name of a schema in which to find the table.

`:table`

Name of a table.

`:column`

Name of a column.

`:role`

Name of a user or group role.

`:privileges`

An array of column privileges the role should be granted to the column.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a single column. The available column privileges are:

- INSERT
- REFERENCES
- SELECT
- UPDATE

Note that column privileges were added in PostgreSQL 8.4, so this function will likley throw an exception on earlier versions.

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT column_privs_are(
    'public', 'frobulate', 'id', 'fred', ARRAY['SELECT', 'UPDATE'],
    'Fred should be able to select and update frobulate.id'
);
SELECT column_privs_are( 'widgets', 'name', 'slim', ARRAY['INSERT', '
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted SELECT on widgets.foo"
#     Extra privileges:
#           INSERT
#           UPDATE
```

Likewise if the role is not granted some of the specified permissions on the table:

```
# Failed test 15: "Role kurk should be granted SELECT, INSERT, UPDATE
#     Missing privileges:
#           INSERT
#           UPDATE
```

In the event that the test fails because the table in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted SELECT on widgets.foo"
#     Table widgets does not exist
```

If the test fails because the column does not actually exist or is not visible, the diagnostics will tell you:

```
# Failed test 17: "Role slim should be granted SELECT on gadgets.foo"
#     Column gadgets.foo does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 18: "Role slim should be granted SELECT on gadgets.foo"
#     Role slim does not exist
```

function_privs_are()

```
SELECT function_privs_are ( :schema, :function, :args, :role, :privil
SELECT function_privs_are ( :schema, :function, :args, :role, :privil
SELECT function_privs_are ( :function, :args, :role, :privileges, :de
SELECT function_privs_are ( :function, :args, :role, :privileges );
```

Parameters

`:schema`

Name of a schema in which to find the function.

`:function`

Name of a function.

`:args`

Array of function arguments.

`:role`

Name of a user or group role.

`:privileges`

An array of function privileges the role should be granted to the function.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a function. The available function privileges are:

- EXECUTE

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT function_privs_are(
    'public', 'frobulate', ARRAY['integer'], 'fred', ARRAY['EXECUTE']
    'Fred should be able to execute frobulate(int)'
);
SELECT function_privs_are( 'bake', '{}', 'slim', '{}');
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted no privileges on foo()"
#     Extra privileges:
#           EXECUTE
```

Likewise if the role is not granted some of the specified permissions on the function:

```
# Failed test 15: "Role kurk should be granted EXECUTE foo()"
#     Missing privileges:
#           EXECUTE
```

In the event that the test fails because the function in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted EXECUTE on foo(int)"
#     Function foo(int) does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted EXECUTE on foo()"
#     Role slim does not exist
```

language_privs_are()

```
SELECT language_privs_are ( :lang, :role, :privileges, :description );
SELECT language_privs_are ( :lang, :role, :privileges );
```

Parameters

`:lang`

Name of a language.

`:role`

Name of a user or group role.

`:privileges`

An array of table privileges the role should be granted to the language.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a language. The available function privileges are:

- USAGE

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT language_privs_are(
    'plpgsql', 'fred', ARRAY['USAGE'],
    'Fred should be granted USAGE on language "flipr"'
);
SELECT language_privs_are( 'plperl', ARRAY['USAGE'] );
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted no privileges on banks"
#     Extra privileges:
#           USAGE
```

Likewise if the role is not granted some of the specified permissions on the language:

```
# Failed test 15: "Role kurk should be granted USAGE on banks"
#     Missing privileges:
#           USAGE
```

In the event that the test fails because the language in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted USAGE on plr"
#     Language plr does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted USAGE on pllolcode"
#     Role slim does not exist
```

`fdw_privs_are()`

```
SELECT fdw_privs_are ( :fdw, :role, :privileges, :description );
SELECT fdw_privs_are ( :fdw, :role, :privileges );
```

Parameters

`:fdw`

Name of a foreign data wrapper.

`:role`

Name of a user or group role.

`:privileges`

An array of table privileges the role should be granted to the foreign data wrapper.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a foreign data wrapper. The available function privileges are:

- USAGE

Note that foreign data wrapper privileges were added in PostgreSQL 8.4, so this function will likley throw an exception on earlier versions.

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT fdw_privs_are(
    'oracle', 'fred', ARRAY['USAGE'],
    'Fred should be granted USAGE on fdw "oracle"'
);
SELECT fdw_privs_are( 'log_csv', ARRAY['USAGE'] );
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted no privileges on odbc"
#     Extra privileges:
#           USAGE
```

Likewise if the role is not granted some of the specified permissions on the fdw:

```
# Failed test 15: "Role kurk should be granted USAGE on odbc"
#     Missing privileges:
#           USAGE
```

In the event that the test fails because the fdw in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted USAGE on FDW sqlite"
#     FDW sqlite does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted USAGE on sqlite"
#     Role slim does not exist
```

server_privs_are()

```
SELECT server_privs_are ( :server, :role, :privileges, :description );
SELECT server_privs_are ( :server, :role, :privileges );
```

Parameters

`:server`

Name of a server.

`:role`

Name of a user or group role.

`:privileges`

An array of table privileges the role should be granted to the server.

`:description`

A short description of the test.

Tests the privileges granted to a role to access a server. The available function privileges are:

- USAGE

Note that server privileges were added in PostgreSQL 8.4, so this function will likley throw an exception on earlier versions.

If the `:description` argument is omitted, an appropriate description will be created. Examples:

```
SELECT server_privs_are(
    'otherdb', 'fred', ARRAY['USAGE'],
    'Fred should be granted USAGE on server "otherdb"'
);
SELECT server_privs_are( 'myserv', ARRAY['USAGE'] );
```

If the role is granted permissions other than those specified, the diagnostics will list the extra permissions, like so:

```
# Failed test 14: "Role bob should be granted no privileges on myserv
#     Extra privileges:
#           USAGE
```

Likewise if the role is not granted some of the specified permissions on the server:

```
# Failed test 15: "Role kurk should be granted USAGE on oltp"
#     Missing privileges:
#           USAGE
```

In the event that the test fails because the server in question does not actually exist or is not visible, you will see an appropriate diagnostic such as:

```
# Failed test 16: "Role slim should be granted USAGE on server oltp"
#     server oltp does not exist
```

If the test fails because the role does not exist, the diagnostics will look something like:

```
# Failed test 17: "Role slim should be granted USAGE on oltp"
#     Role slim does not exist
```

No Test for the Wicked

There is more to pgTAP. Oh *so* much more! You can output your own diagnostics. You can write conditional tests based on the output of utility functions. You can batch up tests in functions. Read on to learn all about it.

Diagnostics

If you pick the right test function, you'll usually get a good idea of what went wrong when it failed. But sometimes it doesn't work out that way. So here we have ways for you to write your own diagnostic messages which are safer than just `\echo` or `SELECT foo`.

diag()

```
SELECT diag( :lines );
```

Parameters

`:lines`

A list of one or more SQL values of the same type.

Returns a diagnostic message which is guaranteed not to interfere with test output. Handy for this sort of thing:

```
-- Output a diagnostic message if the collation is not en_US.UTF-8.
SELECT diag(
     E'These tests expect LC_COLLATE to be en_US.UTF-8,\n',
     'but yours is set to ', setting, E'.\n',
     'As a result, some tests may fail. YMMV.'
)
  FROM pg_settings
```


```
 WHERE name = 'lc_collate'
   AND setting <> 'en_US.UTF-8';
```

Which would produce:

```
# These tests expect LC_COLLATE to be en_US.UTF-8,
# but yours is set to en_US.ISO8859-1.
# As a result, some tests may fail. YMMV.
```

You can pass data of any type to `diag()` on PostgreSQL 8.3 and higher and it will all be converted to text for the diagnostics. On PostgreSQL 8.4 and higher, you can pass any number of arguments (as long as they are all the same data type) and they will be concatenated together.

Conditional Tests

Sometimes running a test under certain conditions will cause the test script or function to die. A certain function or feature isn't implemented (such as `pg_sleep()` prior to PostgreSQL 8.2), some resource isn't available (like a procedural language), or a contrib module isn't available. In these cases it's necessary to skip tests, or declare that they are supposed to fail but will work in the future (a todo test).

skip()

```
SELECT skip( :why, :how_many );
SELECT skip( :how_many, :why );
SELECT skip( :why );
SELECT skip( :how_many );
```

Parameters

`:why`

Reason for skipping the tests.

`:how_many`

Number of tests to skip

Outputs SKIP test results. Use it in a conditional expression within a `SELECT` statement to replace the output of a test that you otherwise would have run.

```
SELECT CASE WHEN pg_version_num() < 80300
    THEN skip('has_enum() not supported before 8.3', 2 )
    ELSE collect_tap(
        has_enum( 'bug_status' ),
```


```
        has_enum( 'bug_status', 'mydesc' )
    ) END;
```

Note how use of the conditional `CASE` statement has been used to determine whether or not to run a couple of tests. If they are to be run, they are run through `collect_tap()`, so that we can run a few tests in the same query. If we don't want to run them, we call `skip()` and tell it how many tests we're skipping.

If you don't specify how many tests to skip, `skip()` will assume that you're skipping only one. This is useful for the simple case, of course:

```
SELECT CASE current_schema()
    WHEN 'public' THEN is( :this, :that )
    ELSE skip( 'Tests not running in the "public" schema' )
    END;
```

But you can also use it in a `SELECT` statement that would otherwise return multiple rows:

```
SELECT CASE current_schema()
    WHEN 'public' THEN is( nspname, 'public' )
    ELSE skip( 'Cannot see the public schema' )
    END
  FROM pg_namespace;
```

This will cause it to skip the same number of rows as would have been tested had the `WHEN` condition been true.

todo()

```
SELECT todo( :why, :how_many );
SELECT todo( :how_many, :why );
SELECT todo( :why );
SELECT todo( :how_many );
```

Parameters

`:why`

Reason for marking tests as to dos.

`:how_many`

Number of tests to mark as to dos.

Declares a series of tests that you expect to fail and why. Perhaps it's because you haven't fixed a bug or haven't finished a new feature:


```
SELECT todo('URIGeller not finished', 2);

\set card '\'Eight of clubs\''
SELECT is( URIGeller.yourCard(), :card, 'Is THIS your card?' );
SELECT is( URIGeller.bendSpoon(), 'bent', 'Spoon bending, how origina
```

With `todo()`, `:how_many` specifies how many tests are expected to fail. If `:how_many` is omitted, it defaults to 1. pgTAP will run the tests normally, but print out special flags indicating they are "todo" tests. The test harness will interpret these failures as ok. Should any todo test pass, the harness will report it as an unexpected success. You then know that the thing you had todo is done and can remove the call to `todo()`.

The nice part about todo tests, as opposed to simply commenting out a block of tests, is that they're like a programmatic todo list. You know how much work is left to be done, you're aware of what bugs there are, and you'll know immediately when they're fixed.

todo_start(why)
todo_start( )

This function allows you declare all subsequent tests as TODO tests, up until the `todo_end()` function is called.

The `todo()` syntax is generally pretty good about figuring out whether or not we're in a TODO test. However, often we find it difficult to specify the *number* of tests that are TODO tests. Thus, you can instead use `todo_start()` and `todo_end()` to more easily define the scope of your TODO tests.

Note that you can nest TODO tests, too:

```
SELECT todo_start('working on this');
-- lots of code
SELECT todo_start('working on that');
-- more code
SELECT todo_end();
SELECT todo_end();
```

This is generally not recommended, but large testing systems often have weird internal needs.

The `todo_start()` and `todo_end()` function should also work with the `todo()` function, although it's not guaranteed and its use is also discouraged:

```
SELECT todo_start('working on this');
-- lots of code
SELECT todo('working on that', 2);
-- Two tests for which the above line applies
-- Followed by more tests scoped till the following line.
SELECT todo_end();
```

We recommend that you pick one style or another of TODO to be on the safe side.

todo_end()

Stops running tests as TODO tests. This function is fatal if called without a preceding `todo_start()` method call.

in_todo()

Returns true if the test is currently inside a TODO block.

Utility Functions

Along with the usual array of testing, planning, and diagnostic functions, pTAP provides a few extra functions to make the work of testing more pleasant.

pgtap_version()

```
SELECT pgtap_version();
```

Returns the version of pgTAP installed in the server. The value is `NUMERIC`, and thus suitable for comparing to a decimal value:

```
SELECT CASE WHEN pgtap_version() < 0.17
    THEN skip('No sequence assertions before pgTAP 0.17')
    ELSE has_sequence('my_big_seq')
    END;
```

pg_version()

```
SELECT pg_version();
```

Returns the server version number against which pgTAP was compiled. This is the stringified version number displayed in the first part of the core `version()` function and stored in the "server_version" setting:


```
try=% select current_setting( 'server_version'), pg_version();
 current_setting | pg_version
-----------------+------------
 8.3.4           | 8.3.4
(1 row)
```

pg_version_num()

```
SELECT pg_version_num();
```

Returns an integer representation of the server version number against which pgTAP was compiled. This function is useful for determining whether or not certain tests should be run or skipped (using `skip()`) depending on the version of PostgreSQL. For example:

```
SELECT CASE WHEN pg_version_num() < 80100
    THEN skip('has_enum() not supported before 8.3' )
    ELSE has_enum( 'bug_status', 'mydesc' )
    END;
```

The revision level is in the tens position, the minor version in the thousands position, and the major version in the ten thousands position and above (assuming PostgreSQL 10 is ever released, it will be in the hundred thousands position). This value is the same as the `server_version_num` setting available in PostgreSQL 8.2 and higher, but supported by this function back to PostgreSQL 8.1:

```
try=% select current_setting( 'server_version_num'), pg_version_num();
 current_setting | pg_version_num
-----------------+----------------
 80304           |          80304
```

os_name()

```
SELECT os_name();
```

Returns a string representing the name of the operating system on which pgTAP was compiled. This can be useful for determining whether or not to skip tests on certain operating systems.

This is usually the same a the output of `uname`, but converted to lower case. There are some semantics in the pgTAP build process to detect other operating systems, though assistance in improving such detection would be greatly appreciated.

NOTE: The values returned by this function may change in the future, depending on how good the pgTAP build process gets at detecting a OS.

collect_tap()

```
SELECT collect_tap(:lines);
```

Parameters

`:lines`

A list of one or more lines of TAP.

Collects the results of one or more pgTAP tests and returns them all. Useful when used in combination with `skip()`:

```
  SELECT CASE os_name() WHEN 'darwin' THEN
      collect_tap(
          cmp_ok( 'Bjørn'::text, '>', 'Bjorn', 'ø > o' ),
          cmp_ok( 'Pınar'::text, '>', 'Pinar', 'ı > i' ),
          cmp_ok( 'José'::text,  '>', 'Jose',  'é > e' ),
          cmp_ok( 'Täp'::text,   '>', 'Tap',   'ä > a' )
      )
  ELSE
       skip('Collation-specific test', 4)
  END;
```

On PostgreSQL 8.4 and higher, it can take any number of arguments. Lower than 8.4 requires the explicit use of an array:

```
SELECT collect_tap(ARRAY[
    ok(true, 'This should pass'),
    ok(false, 'This should fail)
]);
```

display_oper()

```
SELECT display_oper( :opername, :operoid );
```

Parameters

`:opername`

Operator name.

`:operoid`

Operator OID.

Similar to casting an operator OID to regoperator, only the schema is not included in the display. For example:

```
SELECT display_oper(oprname, oid ) FROM pg_operator;
```

Used internally by pgTAP to compare operators, but may be more generally useful.

pg_typeof()

```
SELECT pg_typeof(:any);
```

Parameters

`:any`

Any SQL value.

Returns a `regtype` identifying the type of value passed to the function. This function is used internally by `cmp_ok()` to properly construct types when executing the comparison, but might be generally useful.

```
try=% select pg_typeof(12), pg_typeof(100.2);
 pg_typeof | pg_typeof
-----------+-----------
 integer   | numeric
```

Note: pgTAP does not build `pg_typeof()` on PostgreSQL 8.4 or higher, because it's in core in 8.4. You only need to worry about this if you depend on the function being in particular schema. It will always be in `pg_catalog` in 8.4 and higher.

findfuncs()

```
SELECT findfuncs( :schema, :pattern, :exclude_pattern );
SELECT findfuncs( :schema, :pattern );
SELECT findfuncs( :pattern, :exclude_pattern );
SELECT findfuncs( :pattern );
```

Parameters

`:schema`

Schema to search for functions.

`:pattern`

Regular expression pattern against which to match function names.

`:pattern`

Regular expression pattern to exclude functions with matching names.

This function searches the named schema or, if no schema is passed, the search patch, for all functions that match the regular expression pattern. The optional exclude regular expression pattern can be used to prevent matchin startup/setup/teardown/shutdown functions.

The functions it finds are returned as an array of text values, with each value consisting of the schema name, a dot, and the function name. For example:

```
SELECT findfuncs('tests', '^test);
            findfuncs
----------------------------------
 {tests.test_foo,tests."test bar"}
(1 row)
```

Tap that Batch

Sometimes it can be useful to batch a lot of TAP tests into a function. The simplest way to do so is to define a function that `RETURNS SETOF TEXT` and then simply call `RETURN NEXT` for each TAP test. Here's a simple example:

```
CREATE OR REPLACE FUNCTION my_tests(
) RETURNS SETOF TEXT AS $$
BEGIN
    RETURN NEXT pass( 'plpgsql simple' );
    RETURN NEXT pass( 'plpgsql simple 2' );
END;
$$ LANGUAGE plpgsql;
```

Then you can just call the function to run all of your TAP tests at once:

```
SELECT plan(2);
SELECT * FROM my_tests();
SELECT * FROM finish();
```

do_tap()

```
SELECT do_tap( :schema, :pattern );
SELECT do_tap( :schema );
SELECT do_tap( :pattern );
SELECT do_tap();
```

Parameters

`:schema`

Name of a schema containing pgTAP test functions.

`:pattern`

> Regular expression pattern against which to match function names.

If you like you can create a whole slew of these batched tap functions, and then use the `do_tap()` function to run them all at once. If passed no arguments, it will attempt to find all visible functions that start with "test". If passed a schema name, it will look for and run test functions only in that schema (be sure to cast the schema to `name` if it is the only argument). If passed a regular expression pattern, it will look for function names that match that pattern in the search path. If passed both, it will of course only search for test functions that match the function in the named schema.

This can be very useful if you prefer to keep all of your TAP tests in functions defined in the database. Simply call `plan()`, use `do_tap()` to execute all of your tests, and then call `finish()`. A dead simple example:

```
SELECT plan(32);
SELECT * FROM do_tap('testschema'::name);
SELECT * FROM finish();
```

As a bonus, if `client_min_messages` is set to "warning", "error", "fatal", or "panic", the name of each function will be emitted as a diagnostic message before it is called. For example, if `do_tap()` found and executed two TAP testing functions an `client_min_messages` is set to "warning", output will look something like this:

```
# public.test_this()
ok 1 - simple pass
ok 2 - another simple pass
# public.test_that()
ok 3 - that simple
ok 4 - that simple 2
```

Which will make it much easier to tell what functions need to be examined for failing tests.

runtests()

```
SELECT runtests( :schema, :pattern );
SELECT runtests( :schema );
SELECT runtests( :pattern );
SELECT runtests( );
```

Parameters

`:schema`

Name of a schema containing pgTAP test functions.

`:pattern`

Regular expression pattern against which to match function names.

If you'd like pgTAP to plan, run all of your test functions, and finish, all in one fell swoop, use `runtests()`. This most closely emulates the xUnit testing environment, similar to the functionality of PGUnit and Epic. Example:

```
SELECT * FROM runtests( 'testschema', '^test' );
```

As with `do_tap()`, you can pass in a schema argument and/or a pattern that the names of the tests functions can match. If you pass in only the schema argument, be sure to cast it to `name` to identify it as a schema name rather than a pattern:

```
SELECT * FROM runtests('testschema'::name);
```

Unlike `do_tap()`, `runtests()` fully supports startup, shutdown, setup, and teardown functions, as well as transactional rollbacks between tests. It also outputs the test plan, executes each test function as a TAP subtest, and finishes the tests, so you don't have to call `plan()` or `finish()` yourself. The output, assuming a single startup test, two subtests, and a single shutdown test, will look something like this:

```
ok 1 - Startup test
    # Subtest: public.test_this()
    ok 1 - simple pass
    ok 2 - another simple pass
ok 2 - public.test_this()
    # Subtest: public.test_that()
    ok 1 - that simple
    ok 2 - that simple 2
ok 3 - public.test_that()
ok 4 - Shutdown test
1..4
```

The fixture functions run by `runtests()` are as follows:

- `^startup` - Functions whose names start with "startup" are run in alphabetical order before any test functions are run.

- `^setup` - Functions whose names start with “setup” are run in alphabetical order before each test function is run.
- `^teardown` - Functions whose names start with “teardown” are run in alphabetical order after each test function is run. They will not be run, however, after a test that has died.
- `^shutdown` - Functions whose names start with “shutdown” are run in alphabetical order after all test functions have been run.

Note that all tests executed by `runtests()` are run within a single transaction, and each test is run in a subtransaction that also includes execution all the setup and teardown functions. All transactions are rolled back after each test function, and at the end of testing, leaving your database in largely the same condition as it was in when you started it (the one exception I’m aware of being sequences, which are not rolled back to the value used at the beginning of a rolled-back transaction).

Compose Yourself

So, you’ve been using pgTAP for a while, and now you want to write your own test functions. Go ahead; I don’t mind. In fact, I encourage it. How? Why, by providing a function you can use to test your tests, of course!

But first, a brief primer on writing your own test functions. There isn’t much to it, really. Just write your function to do whatever comparison you want. As long as you have a boolean value indicating whether or not the test passed, you’re golden. Just then use `ok()` to ensure that everything is tracked appropriately by a test script.

For example, say that you wanted to create a function to ensure that two text values always compare case-insensitively. Sure you could do this with `is()` and the `LOWER()` function, but if you’re doing this all the time, you might want to simplify things. Here’s how to go about it:

```
CREATE OR REPLACE FUNCTION lc_is (text, text, text)
RETURNS TEXT AS $$
DECLARE
    result BOOLEAN;
BEGIN
    result := LOWER($1) = LOWER($2);
    RETURN ok( result, $3 ) || CASE WHEN result THEN '' ELSE E'\n' ||
```


```
           '    Have: ' || $1 ||
       E'\n    Want: ' || $2;
) END;
END;
$$ LANGUAGE plpgsql;
```

Yep, that's it. The key is to always use pgTAP's `ok()` function to guarantee that the output is properly formatted, uses the next number in the sequence, and the results are properly recorded in the database for summarization at the end of the test script. You can also provide diagnostics as appropriate; just append them to the output of `ok()` as we've done here.

Of course, you don't have to directly use `ok()`; you can also use another pgTAP function that ultimately calls `ok()`. IOW, while the above example is instructive, this version is easier on the eyes:

```
CREATE OR REPLACE FUNCTION lc_is ( TEXT, TEXT, TEXT )
RETURNS TEXT AS $$
     SELECT is( LOWER($1), LOWER($2), $3);
$$ LANGUAGE sql;
```

But either way, let pgTAP handle recording the test results and formatting the output.

Testing Test Functions

Now you've written your test function. So how do you test it? Why, with this handy-dandy test function!

check_test()

```
SELECT check_test( :test_output, :is_ok, :name, :want_description, :w
SELECT check_test( :test_output, :is_ok, :name, :want_description, :w
SELECT check_test( :test_output, :is_ok, :name, :want_description );
SELECT check_test( :test_output, :is_ok, :name );
SELECT check_test( :test_output, :is_ok );
```

Parameters

`:schema`

Name of a schema containing pgTAP test functions.

`:test_output`

The output from your test. Usually it's just returned by a call to the test function itself. Required.

`:is_ok`

Boolean indicating whether or not the test is expected to pass. Required.

`:name`

A brief name for your test, to make it easier to find failures in your test script. Optional.

`:want_description`

Expected test description to be output by the test. Optional. Use an empty string to test that no description is output.

`:want_diag`

Expected diagnostic message output during the execution of a test. Must always follow whatever is output by the call to `ok()`. Optional. Use an empty string to test that no description is output.

`:match_diag`

Use `matches()` to compare the diagnostics rather than `:is()`. Useful for those situations where you're not sure what will be in the output, but you can match it with a regular expression.

This function runs anywhere between one and three tests against a test function. At its simplest, you just pass in the output of your test function (and it must be one and **only one** test function's output, or you'll screw up the count, so don't do that!) and a boolean value indicating whether or not you expect the test to have passed. That looks something like this:

```
SELECT * FROM check_test(
    lc_eq('This', 'THIS', 'eq'),
    true
);
```

All other arguments are optional, but I recommend that you *always* include a short test name to make it easier to track down failures in your test script. `check_test()` uses this name to construct descriptions of all of the tests it runs. For example, without a short name, the above example will yield output like so:

```
not ok 14 - Test should pass
```

Yeah, but which test? So give it a very succinct name and you'll know what test. If you have a lot of these, it won't be much help. So give each call to `check_test()`

a name:

```
SELECT * FROM check_test(
    lc_eq('This', 'THIS', 'eq'),
    true,
    'Simple lc_eq test',
);
```

Then you'll get output more like this:

```
not ok 14 - Simple lc_test should pass
```

Which will make it much easier to find the failing test in your test script.

The optional fourth argument is the description you expect to be output. This is especially important if your test function generates a description when none is passed to it. You want to make sure that your function generates the test description you think it should! This will cause a second test to be run on your test function. So for something like this:

```
SELECT * FROM check_test(
    lc_eq( ''this'', ''THIS'' ),
    true,
    'lc_eq() test',
    'this is THIS'
);
```

The output then would look something like this, assuming that the `lc_eq()` function generated the proper description (the above example does not):

```
ok 42 - lc_eq() test should pass
ok 43 - lc_eq() test should have the proper description
```

See how there are two tests run for a single call to `check_test()`? Be sure to adjust your plan accordingly. Also note how the test name was used in the descriptions for both tests.

If the test had failed, it would output a nice diagnostics. Internally it just uses `is()` to compare the strings:

```
# Failed test 43:  "lc_eq() test should have the proper description"
#         have: 'this is this'
#         want: 'this is THIS'
```

The fifth argument, `:want_diag`, which is also optional, compares the diagnostics generated during the test to an expected string. Such diagnostics **must** follow whatever is output by the call to `ok()` in your test. Your test function should not call `diag()` until after it calls `ok()` or things will get truly funky.

Assuming you've followed that rule in your `lc_eq()` test function, see what happens when a `lc_eq()` fails. Write your test to test the diagnostics like so:

```
SELECT * FROM check_test(
    lc_eq( ''this'', ''THat'' ),
    false,
    'lc_eq() failing test',
    'this is THat',
    E'    Want: this\n    Have: THat
);
```

This of course triggers a third test to run. The output will look like so:

```
ok 44 - lc_eq() failing test should fail
ok 45 - lc_eq() failing test should have the proper description
ok 46 - lc_eq() failing test should have the proper diagnostics
```

And of course, it the diagnostic test fails, it will output diagnostics just like a description failure would, something like this:

```
# Failed test 46:  "lc_eq() failing test should have the proper diagn
#         have:     Have: this
#     Want: that
#         want:     Have: this
#     Want: THat
```

If you pass in the optional sixth argument, `:match_diag`, the `:want_diag` argument will be compared to the actual diagnostic output using `matches()` instead of `is()`. This allows you to use a regular expression in the `:want_diag` argument to match the output, for those situations where some part of the output might vary, such as time-based diagnostics.

I realize that all of this can be a bit confusing, given the various haves and wants, but it gets the job done. Of course, if your diagnostics use something other than indented "have" and "want", such failures will be easier to read. But either way, *do* test your diagnostics!

Compatibility

Here are some notes on how pgTAP is built for particular versions of PostgreSQL. This helps you to understand any side-effects. If you'd rather not have these changes in your schema, build `pgTAP` with a schema just for it, instead:

```
make TAPSCHEMA=tap
```

To see the specifics for each version of PostgreSQL, consult the files in the `compat/` directory in the pgTAP distribution.

9.3 and Up

No changes. Everything should just work.

9.2 and Down

- Diagnostic output from `lives_ok()` and xUnit function exceptions will not include schema, table, column, data type, or constraint information, since such diagnostics were not introduced until 9.3.

9.1 and Down

- Diagnostic output from `lives_ok()` and xUnit function exceptions will not error context or details, since such diagnostics were not introduced until 9.2.

9.0 and Down

- The `foreign_table_owner_is()` function will not work, because, of course, there were no foreign tables until 9.1.
- The `extensions_are()` functions are not available, because extensions were not introduced until 9.1.

8.4 and Down

- The `sequence_privs_are()` function does not work, because privileges could not be granted on sequences before 9.0.
- The `triggers_are()` function does not ignore internally-created triggers.

8.3 and Down

- A patch is applied to modify `results_eq()` and `row_eq()` to cast records to text before comparing them. This means that things will mainly be correct, but it also means that two queries with incompatible types that convert to the same text string may be considered incorrectly equivalent.
- A C function, `pg_typeof()`, is built and installed in a DSO. This is for compatibility with the same function that ships in 8.4 core, and is required for `cmp_ok()` and `isa_ok()` to work.
- The variadic forms of `diag()` and `collect_tap()` are not available. You can pass an array of TAP to `collect_tap()`, however.
- These permission-testing functions don't work, because one cannot grant permissions on the relevant objects until 8.4:
- `has_any_column_privilege()`
- `has_column_privilege()`
- `has_foreign_data_wrapper_privilege()`
- `has_server_privilege()`

8.2 and Down

- A patch is applied that removes `enum_has_labels()` and `language_owner_is()`, since neither are testable before 8.3.
- the `diag(anyelement)` function and `col_has_default()` cannot be used to test for columns specified with `DEFAULT NULL` (even though that's the implied default default).
- The `has_enums()` function won't work.
- A number of assignments casts are added to increase compatibility. The casts are:
- `boolean` to `text`
- `text[]` to `text`

- `name[]` to `text`
- `regtype` to `text`
- Two operators, `=` and `<>`, are added to compare `name[]` values.

Metadata

Public Repository

The source code for pgTAP is available on GitHub. Please feel free to fork and contribute!

Mail List

Join the pgTAP community by subscribing to the pgtap-users mail list. All questions, comments, suggestions, and bug reports are welcomed there.

Author

David E. Wheeler

Credits

- Michael Schwern and chromatic for Test::More.
- Adrian Howard for Test::Exception.

Copyright and License

DAVID E. WHEELER SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN “AS IS” BASIS, AND DAVID E. WHEELER HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.