nearform / **temporal_tables** Public

Code · Issues 1 · Pull requests 1 · Actions · Security · Insights

master · Code

radomird Add set_system_time function (… ··· on Sep 20 · 74

.github	Bump actions/checkout from 3 t…	last month
scripts	Add set_system_time function (…	last month
test	Add set_system_time function (…	last month
.gitignore	Support history tables with differ…	6 years ago
LICENSE	Update LICENSE	6 years ago
Makefile	Add set_system_time function (…	last month
README.md	Add set_system_time function (…	last month
package.json	chore: release v0.5.0 (#56)	3 months ago
system_time…	Add set_system_time function (…	last month
versioning_f…	Add set_system_time function (…	last month
versioning_f…	Add set_system_time function (…	last month

README.md

Temporal Tables

ci passing

This is an attempt to rewrite the postgresql temporal_tables extension in PL/pgSQL, without the need for external c extension.

The goal is to be able to use it on AWS RDS and other hosted solutions, where using custom extensions or c functions is not an option.

The version provided in `versioning_function.sql` is a drop-in replacement.

About

Postgresql temporal_tables extension in PL/pgSQL, without the need for external c extension.

#hacktoberfest

Readme
View license
Activity
492 stars
106 watching
65 forks
Report repository

Releases 3

v0.5.0 Latest
on Jul 26

+ 2 releases

Contributors 14

+ 3 contributors

Languages

● **PLpgSQL** 87.4% ● **Makefile** 6.3%
● **Shell** 4.4% ● **JavaScript** 1.9%

The version in `versioning_function_nochecks.sql` is similar to the previous one, but all validation checks have been removed. This version is 2x faster than the normal one, but more dangerous and prone to errors.

With time, added some new functionality diverging from the original implementations. New functionalities are however still retro-compatible:

- Ignore updates with no actual changes

Usage

Create a database and the versioning function:

```
createdb temporal_test
psql temporal_test < versioning_function.sql
```

If you would like to have `set_system_time` function available (more details below) you should run the following as well:

```
psql temporal_test < system_time_function.sql
```

Connect to the db:

```
psql temporal_test
```

Create the table to version, in this example it will be a "subscription" table:

```
CREATE TABLE subscriptions
(
  name text NOT NULL,
  state text NOT NULL
);
```

Add the system period column:

```
ALTER TABLE subscriptions
  ADD COLUMN sys_period tstzrange NOT NULL DEFAULT ts
```

Create the history table:

```
CREATE TABLE subscriptions_history (LIKE subscri    or
```

Finally, create the trigger:

```
CREATE TRIGGER versioning_trigger
BEFORE INSERT OR UPDATE OR DELETE ON subscriptions
FOR EACH ROW EXECUTE PROCEDURE versioning(
  'sys_period', 'subscriptions_history', true
);
```

A note on the history table name. Previous versions of this extension quoted and escaped it before usage. Starting version 0.4.0 we are not escaping it anymore and users need to provide the escaped version as a parameter to the trigger.

This is consistent with the c version, simplifies the extension code and fixes an issue with upper case names that weren't properly supported.

Now test with some data:

```
INSERT INTO subscriptions (name, state) VALUES    st
UPDATE subscriptions SET state = 'updated' WHERE name
UPDATE subscriptions SET state = 'updated twice' WHER
DELETE FROM subscriptions WHERE name = 'test1';
```

Take some time between a query and the following, otherwise the difference in the time periods won't be noticeable.

After all the queries are completed, you should check the tables content.

```
SELECT * FROM subscriptions;
```

Should return 0 rows

```
SELECT * FROM subscriptions_history;
```

Should return something similar to:

name	state	sys_period
test1	inserted	["2017-08-01 16:09:45.542983+02","2017-08-01

name	state	sys_period
		16:09:54.984179+02")
test1	updated	["2017-08-01 16:09:54.984179+02","2017-08-01 16:10:08.880571+02")
test1	updated twice	["2017-08-01 16:10:08.880571+02","2017-08-01 16:10:17.33659+02")

Setting custom system time

If you want to take advantage of setting a custom system time you can use the `set_system_time` function. It is a port of the original set_system_time. The function accepts string representation of timestamp in the following format: `YYYY-MM-DD HH:MI:SS.MS.US` - where hours are in 24-hour format 00-23 and the MS (milliseconds) and US (microseconds) portions are optional. Same as the original function, calling it with `null` will reset to default setting (using the CURRENT_TIMESTAMP):

```
SELECT set_system_time(null);
```

Below is an example on how to use this function (continues using the example from above):

Create the set_system_time function:

```
psql temporal_test < system_time_function.sql
```

Set a custom value for the system time:

```
SELECT set_system_time('1999-12-31 23:59:59');
```

Now test with some data:

```
INSERT INTO subscriptions (name, state) VALUES (   st
UPDATE subscriptions SET state = 'updated' WHERE name
UPDATE subscriptions SET state = 'updated twice' WHER
DELETE FROM subscriptions WHERE name = 'test2';
```

Take some time between a query and the following, otherwise the difference in the time periods won't be noticeable.

After all the queries are completed, you should check the `subscriptions_history` table content:

```
SELECT * FROM subscriptions_history;
```

Should return something similar to:

name	state	sys_period
test1	inserted	["2017-08-01 16:09:45.542983+02","2017-08-01 16:09:54.984179+02")
test1	updated	["2017-08-01 16:09:54.984179+02","2017-08-01 16:10:08.880571+02")
test1	updated twice	["2017-08-01 16:10:08.880571+02","2017-08-01 16:10:17.33659+02")
test2	inserted	["1999-12-31 23:59:59+01","1999-12-31 23:59:59.000001+01")
test2	updated	["1999-12-31 23:59:59.000001+01","1999-12-31 23:59:59.000002+01")
test2	updated twice	["1999-12-31 23:59:59.000002+01","1999-12-31 23:59:59.000003+01")

Additional features

Ignore updates without actual change

NOTE: This feature does not work for tables with columns with types that does not support equality operator (e.g. PostGIS types, JSON types, etc.).

By default this extension creates a record in the history table for every update that occurs in the versioned table, regardless of any change actually happening.

We added a fourth paramater to the trigger to change this behaviour and only record updates that result in an actual change.

It is worth mentioning that before making the change, a check is performed on the source table against the history table, in such a way that if the history table has only a subset of the columns of the source table, and you are performing an update in a column that is not present in this subset (this means the column does not exist in the history table), this extension will NOT add a new record to the history. Then you can have columns in the source table that create no new versions if modified by not including those columns in the history table.

The paramater is set by default to `false`, set it to `true` to stop tracking updates without actual changes:

```
CREATE TRIGGER versioning_trigger
BEFORE INSERT OR UPDATE OR DELETE ON subscriptions
FOR EACH ROW EXECUTE PROCEDURE versioning(
  'sys_period', 'subscriptions_history', true, true
);
```

Migrations

During the life of an application is may be necessary to change the schema of a table. In order for temporal_tables to continue to work properly the same migrations should be applied to the history table as well.

What happens if a column is added to the original table but not to the history table?

The new column will be ignored, meaning that the updated row is transferred to the history table, but without the value of the new column. This means that you will lose that specific data.

There are valid use cases for this, for example when you are not interested in storing the historic values of that column.

Beware that temporal_tables won't raise an error

What should I do if I need to remove a column from the original table but want to keep the historic values for it?

You remove the column in the original table, but keep it in the history table - provided it accepts null values.

From that point on the old column in the history table will be ignored and will get null values.

If the column doesn't accept null values you'll need to modify it to allow for null values, otherwise temporal_tables won't be able to create new rows and all operations on the original table will fail

Test

In order to run tests:

```
make run_test
```

The test suite will run the queries in test/sql and store the output in test/result, and will then diff the output from test/result with the prerecorded output in test/expected.

A test suite is also available for the nochecks alternative:

```
make run_test_nochecks
```

Obviously, this suite won't run the tests about the error reporting.

Performance tests

For performance tests run:

```
make performance_test
```

This will create the temporal_tables_test database, add all necessary tables, run test tests and drop the database.

Is it also possible to test against the nochecks version:

```
make performance_test_nochecks
```

or the original c extension run:

```
make performance_test_original
```

This required the original extentions to be installed, but will automatically add it to the database.

On the test machine (my laptop) the complete version is 2x slower than the nochecks versions and 16x slower than the original version.

Two comments about those results:

- original c version makes some use of caching (i.e to share an execution plan), whilst this version doesn't. This is propably accounting for a good chunk of the performance difference. At the moment there's not plan of implementing such caching in this version.
- The trigger still executes in under 1ms and in production environments the the network latency should be more relevant than the trigger itself.

The team

Paolo Chiodi

https://github.com/paolochiodi

https://twitter.com/paolochiodi

Acknowledgements

This project was kindly sponsored by nearForm.

License