PostgreSQL Day Italy 2007

Auditing data changes with tablelog

Andreas 'ads' Scherbaum

http://ads.wars-nicht.de/blog/
PGP: 9F67 73D3 43AA B30E CA8F 56E5 3002 8D24 4813 B5FE

07th Juli 2007

Table of contents

About tablelog

- Trigger-based solution for logging data changes
- Logs go into another table (by default: _log)
- Ability to look back into the row or table history
- License: same as PostgreSQL (BSD)

About tablelog

- Trigger-based solution for logging data changes
- Logs go into another table (by default: _log)
- Ability to look back into the row or table history
- License: same as PostgreSQL (BSD)

About tablelog

- Trigger-based solution for logging data changes
- Logs go into another table (by default: _log)
- Ability to look back into the row or table history
- License: same as PostgreSQL (BSD)

About tablelog

- Trigger-based solution for logging data changes
- Logs go into another table (by default: _log)
- Ability to look back into the row or table history
- License: same as PostgreSQL (BSD)

History

- Started sometime in 2001
- I needed some kind of history for web applications
- Of course the application could write the logs
- But i decided to try it inside the database
- First attempt with PL/pgSQL
- Problem: something like NEW.$column_name is not possible
- So no general solution possible in PL/pgSQL
- tablelog is written in C
- Today PL/Perl or PL/Python would be possible

History

- Started sometime in 2001
- I needed some kind of history for web applications
- Of course the application could write the logs
- But i decided to try it inside the database
- First attempt with PL/pgSQL
- Problem: something like NEW.$column_name is not possible
- So no general solution possible in PL/pgSQL
- tablelog is written in C
- Today PL/Perl or PL/Python would be possible

History

- Started sometime in 2001
- I needed some kind of history for web applications
- Of course the application could write the logs
- But i decided to try it inside the database
- First attempt with PL/pgSQL
- Problem: something like NEW.$column_name is not possible
- So no general solution possible in PL/pgSQL
- tablelog is written in C
- Today PL/Perl or PL/Python would be possible

History

- Started sometime in 2001
- I needed some kind of history for web applications
- Of course the application could write the logs
- But i decided to try it inside the database
- First attempt with PL/pgSQL
- Problem: something like NEW.$column_name is not possible
- So no general solution possible in PL/pgSQL
- tablelog is written in C
- Today PL/Perl or PL/Python would be possible

History

- Started sometime in 2001
- I needed some kind of history for web applications
- Of course the application could write the logs
- But i decided to try it inside the database
- First attempt with PL/pgSQL
- Problem: something like NEW.$column_name is not possible
- So no general solution possible in PL/pgSQL
- tablelog is written in C
- Today PL/Perl or PL/Python would be possible

History

- 2002 first public release
- At this time tablelog was already in use on our servers
- 2002-2005 some small enhancements and bugfixes
- 2005: some helper functions written by Kim Hansen (in PL/pgSQL)
  which help creating a logtable
- 2006: drop support for PG 7.3 and older (OPAQUE -> TRIGGER return value)
- 2007: current release 0.4.4
- Fix compatibility issues with PG 8.2
- RPM and Debian packages

History

- 2002 first public release
- At this time tablelog was already in use on our servers
- 2002-2005 some small enhancements and bugfixes
- 2005: some helper functions written by Kim Hansen (in PL/pgSQL)
  which help creating a logtable
- 2006: drop support for PG 7.3 and older (OPAQUE -> TRIGGER return value)
- 2007: current release 0.4.4
- Fix compatibility issues with PG 8.2
- RPM and Debian packages

History

- 2002 first public release
- At this time tablelog was already in use on our servers
- 2002-2005 some small enhancements and bugfixes
- 2005: some helper functions written by Kim Hansen (in PL/pgSQL)
  which help creating a logtable
- 2006: drop support for PG 7.3 and older (OPAQUE -> TRIGGER return value)
- 2007: current release 0.4.4
- Fix compatibility issues with PG 8.2
- RPM and Debian packages

History

- 2002 first public release
- At this time tablelog was already in use on our servers
- 2002-2005 some small enhancements and bugfixes
- 2005: some helper functions written by Kim Hansen (in PL/pgSQL)
  which help creating a logtable
- 2006: drop support for PG 7.3 and older (OPAQUE -> TRIGGER return value)
- 2007: current release 0.4.4
- Fix compatibility issues with PG 8.2
- RPM and Debian packages

History

- 2002 first public release
- At this time tablelog was already in use on our servers
- 2002-2005 some small enhancements and bugfixes
- 2005: some helper functions written by Kim Hansen (in PL/pgSQL)
  which help creating a logtable
- 2006: drop support for PG 7.3 and older (OPAQUE -> TRIGGER return value)
- 2007: current release 0.4.4
- Fix compatibility issues with PG 8.2
- RPM and Debian packages

Requirements for installation

- root access for installing the library
- Debian or RPM package for your distribution
- NetBSD package soon
- Alternative: source code (latest relase: 0.4.4) and a C compiler
- PostgreSQL header files
- tablelog is on the latest PostgreSQL Live CD

Requirements for installation

- root access for installing the library
- Debian or RPM package for your distribution
- NetBSD package soon
- Alternative: source code (latest relase: 0.4.4) and a C compiler
- PostgreSQL header files
- tablelog is on the latest PostgreSQL Live CD

Requirements for installation

- root access for installing the library
- Debian or RPM package for your distribution
- NetBSD package soon
- Alternative: source code (latest relase: 0.4.4) and a C compiler
- PostgreSQL header files
- tablelog is on the latest PostgreSQL Live CD

Requirements for installation

- root access for installing the library
- Debian or RPM package for your distribution
- NetBSD package soon
- Alternative: source code (latest relase: 0.4.4) and a C compiler
- PostgreSQL header files
- tablelog is on the latest PostgreSQL Live CD

Requirements for installation

- root access for installing the library
- Debian or RPM package for your distribution
- NetBSD package soon
- Alternative: source code (latest relase: 0.4.4) and a C compiler
- PostgreSQL header files
- tablelog is on the latest PostgreSQL Live CD

Installation from source

- Installation from source is easy
- Unpack the source and join the directory
- Type: make USE_PGXS=1
- Type (as root): make USE_PGXS=1 install
- This will install the library into the PostgreSQL library path

Installation from source

- Installation from source is easy
- Unpack the source and join the directory
- Type: make USE_PGXS=1
- Type (as root): make USE_PGXS=1 install
- This will install the library into the PostgreSQL library path

Installation from source

- Installation from source is easy
- Unpack the source and join the directory
- Type: make USE_PGXS=1
- Type (as root): make USE_PGXS=1 install
- This will install the library into the PostgreSQL library path

Installation for one database

In every database you want to use tablelog, you have to create some functions

```
CREATE FUNCTION "table_log" ()
    RETURNS trigger
    AS '\$libdir/table_log', 'table_log' LANGUAGE 'C';
CREATE FUNCTION "table_log_restore_table" (VARCHAR, VARCHAR, CHAR, CHAR, CHAR, TIMESTAMPTZ, CHAR, INT, INT
    RETURNS VARCHAR
    AS '\$libdir/table_log', 'table_log_restore_table' LANGUAGE 'C';
CREATE FUNCTION "table_log_restore_table" (VARCHAR, VARCHAR, CHAR, CHAR, CHAR, TIMESTAMPTZ, CHAR, INT)
    RETURNS VARCHAR
    AS '\$libdir/table_log', 'table_log_restore_table' LANGUAGE 'C';
CREATE FUNCTION "table_log_restore_table" (VARCHAR, VARCHAR, CHAR, CHAR, CHAR, TIMESTAMPTZ, CHAR)
    RETURNS VARCHAR
    AS '\$libdir/table_log', 'table_log_restore_table' LANGUAGE 'C';
CREATE FUNCTION "table_log_restore_table" (VARCHAR, VARCHAR, CHAR, CHAR, CHAR, TIMESTAMPTZ)
    RETURNS VARCHAR
    AS '\$libdir/table_log', 'table_log_restore_table' LANGUAGE 'C';
```

This functions are given for copy&paste in README.table_log
You need to be a database administrator.

Installation for one database

In every database you want to use tablelog, you have to create some functions

```
CREATE FUNCTION "table_log" ()
    RETURNS trigger
    AS '\$libdir/table_log', 'table_log' LANGUAGE 'C';
CREATE FUNCTION "table_log_restore_table" (VARCHAR, VARCHAR, CHAR, CHAR, CHAR, TIMESTAMPTZ, CHAR, INT, INT
    RETURNS VARCHAR
    AS '\$libdir/table_log', 'table_log_restore_table' LANGUAGE 'C';
CREATE FUNCTION "table_log_restore_table" (VARCHAR, VARCHAR, CHAR, CHAR, CHAR, TIMESTAMPTZ, CHAR, INT)
    RETURNS VARCHAR
    AS '\$libdir/table_log', 'table_log_restore_table' LANGUAGE 'C';
CREATE FUNCTION "table_log_restore_table" (VARCHAR, VARCHAR, CHAR, CHAR, CHAR, TIMESTAMPTZ, CHAR)
    RETURNS VARCHAR
    AS '\$libdir/table_log', 'table_log_restore_table' LANGUAGE 'C';
CREATE FUNCTION "table_log_restore_table" (VARCHAR, VARCHAR, CHAR, CHAR, CHAR, TIMESTAMPTZ)
    RETURNS VARCHAR
    AS '\$libdir/table_log', 'table_log_restore_table' LANGUAGE 'C';
```

This functions are given for copy&paste in README.table_log
You need to be a database administrator.

Installing helper functions

- The file table_log_init.sql provides some helper functions
- Usage is optional

Installation with:

```
psql yourdatabase -f table_log_init.sql
```

- You don't need to be a database administrator.
- You need write access to the public schema.
- You need the PL/pgSQL language enabled in your database

This can be done with:

```
postgres@yourhost:/var/lib/postgresql > createlang plpgsql yourdatabase
```

Installing helper functions

- The file table_log_init.sql provides some helper functions
- Usage is optional

Installation with:

```
psql yourdatabase -f table_log_init.sql
```

- You don't need to be a database administrator.
- You need write access to the public schema.
- You need the PL/pgSQL language enabled in your database

This can be done with:

```
postgres@yourhost:/var/lib/postgresql > createlang plpgsql yourdatabase
```

Installing helper functions

- The file table_log_init.sql provides some helper functions
- Usage is optional

Installation with:

```
psql yourdatabase -f table_log_init.sql
```

- You don't need to be a database administrator.
- You need write access to the public schema.
- You need the PL/pgSQL language enabled in your database

This can be done with:

```
postgres@yourhost:/var/lib/postgresql > createlang plpgsql yourdatabase
```

Create a logging table

```
tablelog=# CREATE TABLE test (id SERIAL, content VARCHAR);
NOTICE:  CREATE TABLE will create implicit sequence "test_id_seq" for serial column "test.id"
CREATE TABLE
tablelog=# SELECT * FROM test;
 id | content
----+---------
(0 rows)
```

```
tablelog=# SELECT table_log_init(5, 'test');
NOTICE:  CREATE TABLE will create implicit sequence "test_log_trigger_id_seq" for serial column "test_log.
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test_log_pkey" for table "test_log"
 table_log_init
----------------

(1 row)
tablelog=# SELECT * FROM test_log;
 id | content | trigger_mode | trigger_tuple | trigger_changed | trigger_id | trigger_user
----+---------+--------------+---------------+-----------------+------------+--------------
(0 rows)
```

Create a logging table

```
tablelog=# CREATE TABLE test (id SERIAL, content VARCHAR);
NOTICE:  CREATE TABLE will create implicit sequence "test_id_seq" for serial column "test.id"
CREATE TABLE
tablelog=# SELECT * FROM test;
 id | content
----+---------
(0 rows)


tablelog=# SELECT table_log_init(5, 'test');
NOTICE:  CREATE TABLE will create implicit sequence "test_log_trigger_id_seq" for serial column "test_log.
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test_log_pkey" for table "test_log"
 table_log_init
----------------

(1 row)
tablelog=# SELECT * FROM test_log;
 id | content | trigger_mode | trigger_tuple | trigger_changed | trigger_id | trigger_user
----+---------+--------------+---------------+-----------------+------------+--------------
(0 rows)
```

Extra columns in the logging table

- tablelog is using the same table structure like the original table but without constraints
- tablelog adds 3-5 extra columns
- Additional columns:
- trigger_mode: contains 'INSERT', 'UPDATE' or 'DELETE'
- trigger_tuple: contains 'old' or 'new'
- trigger_changed: timestamp when the trigger was called
- Additional but optional columns:
- trigger_user: contains the session username (the one who connected to the database)
- trigger_id: BIGSERIAL as primary key for the logging table

Extra columns in the logging table

- tablelog is using the same table structure like the original table but without constraints
- tablelog adds 3-5 extra columns
- Additional columns:
- trigger_mode: contains 'INSERT', 'UPDATE' or 'DELETE'
- trigger_tuple: contains 'old' or 'new'
- trigger_changed: timestamp when the trigger was called
- Additional but optional columns:
- trigger_user: contains the session username (the one who connected to the database)
- trigger_id: BIGSERIAL as primary key for the logging table

Extra columns in the logging table

- tablelog is using the same table structure like the original table but without constraints
- tablelog adds 3-5 extra columns
- Additional columns:
- trigger_mode: contains 'INSERT', 'UPDATE' or 'DELETE'
- trigger_tuple: contains 'old' or 'new'
- trigger_changed: timestamp when the trigger was called
- Additional but optional columns:
- trigger_user: contains the session username (the one who connected to the database)
- trigger_id: BIGSERIAL as primary key for the logging table

Extra columns in the logging table

- tablelog is using the same table structure like the original table but without constraints
- tablelog adds 3-5 extra columns
- Additional columns:
- trigger_mode: contains 'INSERT', 'UPDATE' or 'DELETE'
- trigger_tuple: contains 'old' or 'new'
- trigger_changed: timestamp when the trigger was called
- Additional but optional columns:
- trigger_user: contains the session username (the one who connected to the database)
- trigger_id: BIGSERIAL as primary key for the logging table

Extra columns in the logging table

- tablelog is using the same table structure like the original table but without constraints
- tablelog adds 3-5 extra columns
- Additional columns:
- trigger_mode: contains 'INSERT', 'UPDATE' or 'DELETE'
- trigger_tuple: contains 'old' or 'new'
- trigger_changed: timestamp when the trigger was called
- Additional but optional columns:
- trigger_user: contains the session username (the one who connected to the database)
- trigger_id: BIGSERIAL as primary key for the logging table

Extra columns in the logging table

- tablelog is using the same table structure like the original table but without constraints
- tablelog adds 3-5 extra columns
- Additional columns:
- trigger_mode: contains 'INSERT', 'UPDATE' or 'DELETE'
- trigger_tuple: contains 'old' or 'new'
- trigger_changed: timestamp when the trigger was called
- Additional but optional columns:
- trigger_user: contains the session username (the one who connected to the database)
- trigger_id: BIGSERIAL as primary key for the logging table

Extra columns in the logging table

- tablelog is using the same table structure like the original table but without constraints
- tablelog adds 3-5 extra columns
- Additional columns:
- trigger_mode: contains 'INSERT', 'UPDATE' or 'DELETE'
- trigger_tuple: contains 'old' or 'new'
- trigger_changed: timestamp when the trigger was called
- Additional but optional columns:
- trigger_user: contains the session username (the one who connected to the database)
- trigger_id: BIGSERIAL as primary key for the logging table

What does tablelog exactly do?

Insert data:

```
tablelog=# INSERT INTO test (content) VALUES ('string 1');
INSERT 0 1
tablelog=# SELECT * FROM test; SELECT * FROM test_log;
 id | content
----+----------
  1 | string 1
(1 row)

 id | content  | trigger_mode | trigger_tuple |        trigger_changed         | trigger_id | trigger_user
----+----------+--------------+---------------+--------------------------------+------------+--------------
  1 | string 1 | INSERT       | new           | 2007-07-07 09:45:46.019119+02 |          1 | ads
(1 row)
```

What does tablelog exactly do?

Update data:

```
tablelog=# UPDATE test SET content='string 2' WHERE content='string 1';
UPDATE 1
tablelog=# SELECT * FROM test; SELECT * FROM test_log ORDER BY trigger_id;
 id | content
----+----------
  1 | string 2
(1 row)

 id | content  | trigger_mode | trigger_tuple |        trigger_changed        | trigger_id | trigger_user
----+----------+--------------+---------------+-------------------------------+------------+--------------
  1 | string 1 | INSERT       | new           | 2007-07-07 09:45:46.019119+02 |          1 | ads
  1 | string 1 | UPDATE       | old           | 2007-07-07 09:47:40.743719+02 |          2 | ads
  1 | string 2 | UPDATE       | new           | 2007-07-07 09:47:40.743719+02 |          3 | ads
(3 rows)
```

What does tablelog exactly do?

Delete data:

```
tablelog=# DELETE FROM test WHERE content='string 2';
DELETE 1
tablelog=# SELECT * FROM test; SELECT * FROM test_log ORDER BY trigger_id;
 id | content
----+---------
(0 rows)

 id | content  | trigger_mode | trigger_tuple |        trigger_changed         | trigger_id | trigger_user
----+----------+--------------+---------------+--------------------------------+------------+--------------
  1 | string 1 | INSERT       | new           | 2007-07-07 09:45:46.019119+02  |          1 | ads
  1 | string 1 | UPDATE       | old           | 2007-07-07 09:47:40.743719+02  |          2 | ads
  1 | string 2 | UPDATE       | new           | 2007-07-07 09:47:40.743719+02  |          3 | ads
  1 | string 2 | DELETE       | old           | 2007-07-07 09:49:44.62334+02   |          4 | ads
(4 rows)
```

Looking into the history

Table state right now:

```
tablelog=# DELETE FROM test; DELETE FROM test_log;
DELETE 0
DELETE 4
tablelog=# INSERT INTO test (content) VALUES ('string 1');
INSERT 0 1
tablelog=# SELECT id, content FROM
tablelog-#   (SELECT DISTINCT ON (id) id, content, trigger_tuple
tablelog(#                    FROM test_log
tablelog(#                   WHERE trigger_changed <= NOW()
tablelog(#                ORDER BY id DESC, trigger_id DESC) AS tablelog1
tablelog-#  WHERE trigger_tuple='new' ORDER BY id;
 id | content
----+----------
  2 | string 1
(1 row)
```

Looking into the history

Table state some time ago:

```
tablelog=# UPDATE test SET content='string 2' WHERE content='string 1';
UPDATE 1
tablelog=# SELECT * FROM test_log ORDER BY trigger_id;
 id | content  | trigger_mode | trigger_tuple |        trigger_changed         | trigger_id | trigger_user
----+----------+--------------+---------------+--------------------------------+------------+--------------
  2 | string 1 | INSERT       | new           | 2007-07-07 10:00:26.703507+02 |          5 | ads
  2 | string 1 | UPDATE       | old           | 2007-07-07 10:01:13.766625+02 |          6 | ads
  2 | string 2 | UPDATE       | new           | 2007-07-07 10:01:13.766625+02 |          7 | ads
(3 rows)
```

```
tablelog=# SELECT id, content FROM
tablelog-#   (SELECT DISTINCT ON (id) id, content, trigger_tuple
tablelog(#                  FROM test_log
tablelog(#                 WHERE trigger_changed <= '2007-07-07 10:00:30.000000+02'
tablelog(#              ORDER BY id DESC, trigger_id DESC) AS tablelog1
tablelog-#  WHERE trigger_tuple='new' ORDER BY id;
 id | content
----+----------
  2 | string 1
(1 row)
```

Looking into the history

Table state some time ago:

```
tablelog=# UPDATE test SET content='string 2' WHERE content='string 1';
UPDATE 1
tablelog=# SELECT * FROM test_log ORDER BY trigger_id;
 id | content  | trigger_mode | trigger_tuple |        trigger_changed         | trigger_id | trigger_user
----+----------+--------------+---------------+--------------------------------+------------+--------------
  2 | string 1 | INSERT       | new           | 2007-07-07 10:00:26.703507+02 |          5 | ads
  2 | string 1 | UPDATE       | old           | 2007-07-07 10:01:13.766625+02 |          6 | ads
  2 | string 2 | UPDATE       | new           | 2007-07-07 10:01:13.766625+02 |          7 | ads
(3 rows)


tablelog=# SELECT id, content FROM
tablelog-#   (SELECT DISTINCT ON (id) id, content, trigger_tuple
tablelog(#                   FROM test_log
tablelog(#                  WHERE trigger_changed <= '2007-07-07 10:00:30.000000+02'
tablelog(#               ORDER BY id DESC, trigger_id DESC) AS tablelog1
tablelog-#  WHERE trigger_tuple='new' ORDER BY id;
 id | content
----+----------
  2 | string 1
(1 row)
```

Looking into the history

Table state some time ago:

```
tablelog=# DELETE FROM test WHERE content='string 2';
DELETE 1
tablelog=# SELECT * FROM test; SELECT * FROM test_log ORDER BY trigger_id;
 id | content
----+---------
(0 rows)

 id | content  | trigger_mode | trigger_tuple |        trigger_changed         | trigger_id | trigger_user
----+----------+--------------+---------------+--------------------------------+------------+--------------
  2 | string 1 | INSERT       | new           | 2007-07-07 10:00:26.703507+02 |          5 | ads
  2 | string 1 | UPDATE       | old           | 2007-07-07 10:01:13.766625+02 |          6 | ads
  2 | string 2 | UPDATE       | new           | 2007-07-07 10:01:13.766625+02 |          7 | ads
  2 | string 2 | DELETE       | old           | 2007-07-07 10:05:46.098647+02 |          8 | ads
(4 rows)

tablelog=# SELECT id, content FROM
tablelog-#   (SELECT DISTINCT ON (id) id, content, trigger_tuple
tablelog(#                  FROM test_log
tablelog(#                 WHERE trigger_changed <= '2007-07-07 10:02:00.000000+02'
tablelog(#              ORDER BY id DESC, trigger_id DESC) AS tablelog1
tablelog-#  WHERE trigger_tuple='new' ORDER BY id;
 id | content
----+----------
  2 | string 2
(1 row)
```

Looking into the history

Table state some time ago:

```
tablelog=# DELETE FROM test WHERE content='string 2';
DELETE 1
tablelog=# SELECT * FROM test; SELECT * FROM test_log ORDER BY trigger_id;
 id | content
----+---------
(0 rows)

 id | content   | trigger_mode | trigger_tuple |        trigger_changed         | trigger_id | trigger_user
----+----------+--------------+---------------+--------------------------------+------------+--------------
  2 | string 1 | INSERT       | new           | 2007-07-07 10:00:26.703507+02 |          5 | ads
  2 | string 1 | UPDATE       | old           | 2007-07-07 10:01:13.766625+02 |          6 | ads
  2 | string 2 | UPDATE       | new           | 2007-07-07 10:01:13.766625+02 |          7 | ads
  2 | string 2 | DELETE       | old           | 2007-07-07 10:05:46.098647+02 |          8 | ads
(4 rows)

tablelog=# SELECT id, content FROM
tablelog-#   (SELECT DISTINCT ON (id) id, content, trigger_tuple
tablelog(#                  FROM test_log
tablelog(#                 WHERE trigger_changed <= '2007-07-07 10:02:00.000000+02'
tablelog(#              ORDER BY id DESC, trigger_id DESC) AS tablelog1
tablelog-#  WHERE trigger_tuple='new' ORDER BY id;
 id | content
----+----------
  2 | string 2
(1 row)
```

Create temporary table from results

```
tablelog=# SELECT table_log_restore_table('test', 'id', 'test_log', 'trigger_id',
tablelog(#                                'show_log', '2007-07-07 10:02:00.000000+02',
tablelog(#                                NULL, 0, 0);
 table_log_restore_table
-------------------------
 show_log
(1 row)
```

```
tablelog=# SELECT * FROM show_log;
 id | content
----+----------
  2 | string 2
(1 row)
```

- 7. param: value (from primary key) to restore
- 8. param: 0 - restore forward, 1 - restore backwards
- 9. param: 0 - create temporary output table, 1 - create normal output table, default: temporary table

Create temporary table from results

```
tablelog=# SELECT table_log_restore_table('test', 'id', 'test_log', 'trigger_id',
tablelog(#                                'show_log', '2007-07-07 10:02:00.000000+02',
tablelog(#                                NULL, 0, 0);
 table_log_restore_table
-------------------------
 show_log
(1 row)


tablelog=# SELECT * FROM show_log;
 id | content
----+----------
  2 | string 2
(1 row)
```

- 7. param: value (from primary key) to restore
- 8. param: 0 - restore forward, 1 - restore backwards
- 9. param: 0 - create temporary output table, 1 - create normal output table, default: temporary table

Create temporary table from results

```
tablelog=# SELECT table_log_restore_table('test', 'id', 'test_log', 'trigger_id',
tablelog(#                                'show_log', '2007-07-07 10:02:00.000000+02',
tablelog(#                                NULL, 0, 0);
 table_log_restore_table
-------------------------
 show_log
(1 row)


tablelog=# SELECT * FROM show_log;
 id | content
----+----------
  2 | string 2
(1 row)
```

- 7. param: value (from primary key) to restore
- 8. param: 0 - restore forward, 1 - restore backwards
- 9. param: 0 - create temporary output table, 1 - create normal output table, default: temporary table

Requirements for an audit

- Check permissions on your tables
- Logging table should have SELECT permissions at max for any normal user
- Use different users for normal operations and for auditing
- Create the table_log functions with 'SECURITY DEFINER' privileges
- Existing functions can be modified with:
  ALTER FUNCTION table_log() SECURITY DEFINER;
- Result: user can change the audited table in any way
- You have a full backlog of what was changed

Requirements for an audit

- Check permissions on your tables
- Logging table should have SELECT permissions at max for any normal user
- Use different users for normal operations and for auditing
- Create the table_log functions with 'SECURITY DEFINER' privileges
- Existing functions can be modified with:
  ALTER FUNCTION table_log() SECURITY DEFINER;
- Result: user can change the audited table in any way
- You have a full backlog of what was changed

Requirements for an audit

- Check permissions on your tables
- Logging table should have SELECT permissions at max for any normal user
- Use different users for normal operations and for auditing
- Create the table_log functions with 'SECURITY DEFINER' privileges
- Existing functions can be modified with:
  ALTER FUNCTION table_log() SECURITY DEFINER;
- Result: user can change the audited table in any way
- You have a full backlog of what was changed

Requirements for an audit

- Check permissions on your tables
- Logging table should have SELECT permissions at max for any normal user
- Use different users for normal operations and for auditing
- Create the table_log functions with 'SECURITY DEFINER' privileges
- Existing functions can be modified with:
  ALTER FUNCTION table_log() SECURITY DEFINER;
- Result: user can change the audited table in any way
- You have a full backlog of what was changed

Drawbacks

- Biggest drawback: tablelog is using NOW() and so gets the timestamp when the transaction starts
- timeofday() would be no better solution
- tablelog would need the commit timestamp when the data became visible
- There's some overhead because all data is written at least twice
- As example: the current data is also available in the original table but it's much easier to get the full history with one simple query

Drawbacks

- Biggest drawback: tablelog is using NOW() and so gets the timestamp when the transaction starts
- timeofday() would be no better solution
- tablelog would need the commit timestamp when the data became visible
- There's some overhead because all data is written at least twice
- As example: the current data is also available in the original table but it's much easier to get the full history with one simple query

Drawbacks

- Biggest drawback: tablelog is using NOW() and so gets the timestamp when the transaction starts
- timeofday() would be no better solution
- tablelog would need the commit timestamp when the data became visible
- There's some overhead because all data is written at least twice
- As example: the current data is also available in the original table but it's much easier to get the full history with one simple query

Drawbacks

- Biggest drawback: tablelog is using NOW() and so gets the timestamp when the transaction starts
- timeofday() would be no better solution
- tablelog would need the commit timestamp when the data became visible
- There's some overhead because all data is written at least twice
- As example: the current data is also available in the original table but it's much easier to get the full history with one simple query

Drawbacks

- Biggest drawback: tablelog is using NOW() and so gets the timestamp when the transaction starts
- timeofday() would be no better solution
- tablelog would need the commit timestamp when the data became visible
- There's some overhead because all data is written at least twice
- As example: the current data is also available in the original table but it's much easier to get the full history with one simple query

Drawbacks

- The current implementation is adding too much overhead in trigger_mode and trigger_tuple
- 'INSERT' could be 'I', UPDATE could be 'U', DELETE could be 'D'
- 'old' and 'new' could even be a boolean
- This will be changed in 1.0.0

Drawbacks

- The current implementation is adding too much overhead in trigger_mode and trigger_tuple
- 'INSERT' could be 'I', UPDATE could be 'U', DELETE could be 'D'
- 'old' and 'new' could even be a boolean
- This will be changed in 1.0.0

Availability - Where to get tablelog

- Source: `http://pgfoundry.org/projects/tablelog/`
- RPM: `http://developer.postgresql.org/~devrim/rpms/other/postgresql-table_log/`
- In Fedora Core
- .deb src: `http://base.wars-nicht.de/projects/Pg/`
- In Debian unstable

Thanks

- Thanks to:
- Kim Hansen for providing the helper functions
- Devrim Gunduz for providing RPM packages
- Alexander Wirt for help creating Debian packages
- Robert Bernier for creating the pg_live CD
- PostgreSQL community for answering my questions
- All the other people who provided ideas or small bugfixes

Thanks

- Thanks to:
- Kim Hansen for providing the helper functions
- Devrim Gunduz for providing RPM packages
- Alexander Wirt for help creating Debian packages
- Robert Bernier for creating the pg_live CD
- PostgreSQL community for answering my questions
- All the other people who provided ideas or small bugfixes

Thanks

- Thanks to:
- Kim Hansen for providing the helper functions
- Devrim Gunduz for providing RPM packages
- Alexander Wirt for help creating Debian packages
- Robert Bernier for creating the pg_live CD
- PostgreSQL community for answering my questions
- All the other people who provided ideas or small bugfixes

End

http://pgfoundry.org/projects/tablelog/

Questions?

Andreas 'ads' Scherbaum <andreas@scherbaum.biz>
PostgreSQL User Group Germany