Event Triggers
A.K.A The Real Mess.

Dimitri Fontaine dimitri@2ndQuadrant.fr

February, 3rd 2013
Dimitri Fontaine

2ndQuadrant France
PostgreSQL Major Contributor

- pgloader, prefix, skytools, debian, ...
- CREATE EXTENSION
- CREATE EVENT TRIGGER
- Bi-Directional Réplication
- Partitionning
pgloader, prefix, skytools, debian, ...

CREATE EXTENSION

CREATE EVENT TRIGGER

Bi-Directional Réplication

Partitionning
pgloader, prefix, skytools, debian, ...
CREATE EXTENSION
CREATE EVENT TRIGGER
Bi-Directional Réplication
Partitionning
2ndQuadrant France
PostgreSQL Major Contributor

- pgloader, prefix, skytools, debian, ...
- CREATE EXTENSION
- CREATE EVENT TRIGGER
- *Bi-Directional Réplication*
- Partitionning
So, **Event Triggers**, what do you mean?
It always starts *simple*

```sql
create table foo(a text, b int);

select a, b
  from relation r
where a > '2013'
```
It always starts *simple enough*

create table foo(a int, b int);

select a, b
  from relation r
where a > '2013'
It always starts *simple*... then we try handling *time*

create table foo(a date, b int);

```sql
select a, b
  from relation r
where a > '2013'
```
create extension hstore;

cREATE TABLE testhstore (h hstore);

SELECT count(*)
  FROM testhstore
WHERE h @> 'wait=>CC, public=>t';
PostgreSQL supports Extensions

Data Type Specific Indexing and Query Support

- Functions, Aggregates, Window Functions
- Data Types with Input/Output functions
- Casts (implicit, assignment only)
- Operators
- Operator Class, Operator Family
- *and more...*
With PostgreSQL you can tweak INSERT, UPDATE, DELETE

- Maintain a Materialized View
- Apply crossing threshold discounts
  - Trigger external actions on some events
  - NOTIFY some other application parts (e.g. cache)
- Queue events to process later (Use PGQ)
- Replicate data (Slony, Londiste, Bucardo...)
With PostgreSQL you can tweak INSERT, UPDATE, DELETE

- Maintain a Materialized View
- Apply crossing threshold discounts
- Trigger external actions on some events
- NOTIFY some other application parts (e.g. cache)
- Queue events to process later (Use PGQ)
- Replicate data (Slony, Londiste, Bucardo...)

Dimitri Fontaine dimitri@2ndQuadrant.fr

Event Triggers

February, 3rd 2013
With PostgreSQL you can tweak INSERT, UPDATE, DELETE

- Maintain a Materialized View
- Apply crossing threshold discounts
- Trigger external actions on some events
- NOTIFY some other application parts (e.g. cache)
- Queue events to process later (Use PGQ)
- Replicate data (Slony, Londiste, Bucardo...)

Dimitri Fontaine

February, 3rd 2013
CREATE TABLE main_table (a int, b int);

CREATE FUNCTION trigger_func()
    RETURNS trigger
    LANGUAGE plpgsql AS 'BEGIN
        RAISE NOTICE ''trigger_func(%) called: action = %, when = %, TG_ARGV[0], TG_OP, TG_WHEN, TG_LEVEL;
        RETURN NULL;
    END;';
CREATE TRIGGER before_ins_stmt_trig
BEFORE INSERT ON main_table
FOR EACH STATEMENT
EXECUTE PROCEDURE trigger_func('before_ins_stmt');

CREATE TRIGGER after_ins_stmt_trig
AFTER INSERT ON main_table
FOR EACH STATEMENT
EXECUTE PROCEDURE trigger_func('after_ins_stmt');
create table foo
  (id serial primary key,
   f1 text)
);

alter table foo
  add column f2 text check (upper(f2) = f2);
Here's what foo looks like now

```sql
\d foo
```

Table "public.foo"

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td>not null default nextval('foo_id_seq'::regclass)</td>
</tr>
<tr>
<td>f1</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td>text</td>
<td></td>
</tr>
</tbody>
</table>

Indexes:

- "foo_pkey" PRIMARY KEY, btree (id)

Check constraints:

- "foo_f2_check" CHECK (upper(f2) = f2)
What if you could *tweak* DDL too

Some **DDL** Trigger use cases

- Audit trail
- Replication Triggers
- Implement Local Policies
- Divert Execution
- Limited Granting of DDL privileges with Security Definer trigger functions
What if you could tweak DDL too

Some **Event** Trigger use cases

- `sql_drop (CASCADE)`
- Prevent Table Rewrite (except at night) (unless full moon)
- Create Table If Not Exists, at `INSERT` time
- Integrated Extension Package Management
What if you could *tweak* DDL too

Some **Event** Trigger use cases

- `sql_drop (CASCADE)`
- Prevent Table Rewrite (except at night) (unless full moon)
- Create Table If Not Exists, at **INSERT** time
- Integrated Extension Package Management
What if you could *tweak* DDL too

Some **Event** Trigger use cases

- `sql_drop` (CASCADE)
- Prevent Table Rewrite (except at night) (unless full moon)
- Create Table If Not Exists, at INSERT time
- Integrated Extension Package Management
What if you could *tweak* DDL too

Some **Event** Trigger use cases

- `sql_drop (CASCADE)`
- Prevent Table Rewrite (except at night) (unless full moon)
- Create Table If Not Exists, at INSERT time
- Integrated Extension Package Management
CREATE OR REPLACE FUNCTION abort_any_command()
RETURNS event_trigger
LANGUAGE plpgsql
AS 'BEGIN
  RAISE EXCEPTION ''command % is disabled'', tg_tag;
END;
';

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
  EXECUTE PROCEDURE abort_any_command();
Of course, the usual ALTER and DROP commands

```
ALTER EVENT TRIGGER abort_ddl DISABLE;
ALTER EVENT TRIGGER abort_ddl ENABLE replica|always;
ALTER EVENT TRIGGER abort_ddl OWNER TO bob;
ALTER EVENT TRIGGER abort_ddl RENAME TO assimilated;
DROP EVENT TRIGGER abort_ddl;
```
Limited Number of Events Supported now

- `ddl_command_start`
- `ddl_command_end`
- `sql_drop` *currently in review*
~# create table bar(a int, b int);
CREATE TABLE

~# create function add1(int) returns int
   language sql as 'select \$1+1';
CREATE FUNCTION

~# drop function add1(int);
DROP FUNCTION
create function test_event_trigger()
  returns event_trigger as '
BEGIN
  RAISE NOTICE ''test_event_trigger: % %’’, tg_event, tg_tag;
END
' language plpgsql;

create function test_event_trigger_drop_function()
  returns event_trigger as '
BEGIN
  drop function test_event_trigger() cascade;
END
' language plpgsql;
And now let’s have some fun

```sql
create event trigger drop_test_b on "ddl_command_start"
    execute procedure test_event_trigger();

create event trigger drop_test_a on "ddl_command_start"
    when tag in ('create table')
    execute procedure test_event_trigger_drop_function();

create table event_trigger_fire1 (a int);
```
Event Triggers Information

Currently given as magic variables available in PL/pgSQL

We have

- TG_EVENT
- TG_TAG

We want to add

- TG_OPERATION
- TG_OBTYPE_NAME
- TG_OBJECT_ID
- TG_OBJECT_NAME
- TG_SCHEMA_NAME
What about generated commands?

Current proposal is TG_CONTEXT. See the worked out tracking examples at http://www.postgresql.org/message-id/m2han7xyzp.fsf@2ndQuadrant.fr

```sql
create event trigger track_table on ddl_command_trace
    when tag in ('create table', 'alter table', 'drop table')
    and context in ('toplevel', 'generated', 'subcommand')
execute procedure public.track_table_activity();
```
And still some more

create schema baz
  authorization dim

create table distributors
  (did serial primary key,
   name varchar(40),
   f2 text check (upper(f2) = f2),
   unique(name) with (fillfactor=70)
  )
with (fillfactor=70);
NOTICE:  snitch event: ddl_command_end, context: GENERATED, tag: CREATE SEQUENCE, operation: CREATE, type: SEQUENCE
NOTICE:  oid: 41633, schema: baz, name: distributors_did_seq
NOTICE:  command: CREATE SEQUENCE baz.distributors_did_seq;

NOTICE:  snitch event: ddl_command_end, context: SUBCOMMAND, tag: CREATE TABLE, operation: CREATE, type: TABLE
NOTICE:  oid: 41635, schema: baz, name: distributors
NOTICE:  command: CREATE TABLE baz.distributors
did integer,
   name pg_catalog.varchar,
f2 text,
CHECK ((upper(f2) = f2))) WITH (fillfactor=70);
NOTICE: snitch event: ddl_command_end, context: GENERATED, tag: CREATE INDEX, operation: CREATE, type: INDEX
NOTICE: oid: 41643, schema: baz, name: distributors_pkey
NOTICE: command: CREATE UNIQUE INDEX distributors_pkey
ON baz.distributors USING btree (did);

NOTICE: snitch event: ddl_command_end, context: GENERATED, tag: CREATE INDEX, operation: CREATE, type: INDEX
NOTICE: oid: 41645, schema: baz, name: distributors_name_key
NOTICE: command: CREATE UNIQUE INDEX distributors_name_key
ON baz.distributors USING btree (name)
WITH (fillfactor=70);
NOTICE: snitch event: ddl_command_end, context: GENERATED, tag: ALTER SEQUENCE, operation: ALTER, type: SEQUENCE
NOTICE: oid: 41633, schema: baz, name: distributors_did_seq
NOTICE: command: ALTER SEQUENCE baz.distributors_did_seq
       OWNED BY baz.distributors.did;

NOTICE: snitch event: ddl_command_end, context: TOPLEVEL, tag: CREATE SCHEMA, operation: CREATE, type: SCHEMA
NOTICE: oid: 41632, schema: <NULL>, name: baz
NOTICE: command: CREATE SCHEMA baz AUTHORIZATION dim;

CREATE SCHEMA
How to get at *generated* commands?

**PRO**
- consider them DDLs
- `ProcessUtility()`
- `ProcessUtilityContext`

**CONS**
- the user didn’t type a command
- clean up the code
- it’s another kind of event
Next features

Some features are still on the *todo* list

- **INSTEAD OF**
  - `table rewrite`
  - `create table on insert`
  - `add column on update`
Next features

Some features are still on the todo list

- INSTEAD OF
- *table rewrite*
  - *create table on insert*
  - *add column on update*
Next features

Some features are still on the *todo* list

- INSTEAD OF
- *table rewrite*
- *create table on insert*
- *add column on update*
Next features

Some features are still on the *todo* list

- **INSTEAD OF**
- *table rewrite*
- *create table on insert*
- *add column on update*
Instead Of Event Triggers 1/2

```sql
create event trigger my_create_extension
instead of 'create extension'
execute procedure my_create_extension();
```
create function my_create_extension()
    returns event_trigger
    language plpgsql
as ' 
begin
    alter event trigger my_create_extension disable;
    -- do some stuff here
    create extension tg_objectid;
    -- do some more stuff here, presumably
end;
';
Conclusion

Any Question? Now is the time to ask!