3.33pt

Capturing DDL events

... and a JSON representation for them

Álvaro Herrera
alvherre@2ndQuadrant.com

2ndQuadrant Ltd.
http://www.2ndQuadrant.com/

PGBR 2015

http://pgbr.postgresql.org.br/2015/

2ndQuadrant
Professional PostgreSQL

What is DDL?

- “Data Definition Language”
- Stuff most people are familiar with

What is DDL?

- “Data Definition Language”
- Stuff most people are familiar with
- on this room anyway
  - `CREATE { TABLE , FUNCTION , VIEW , ... }`
    - 36 CREATE commands
  - `ALTER { TABLE, OPERATOR FAMILY, ... }`
    - 37 ALTER commands
    - 43 sub-commands ALTER TABLE
  - `DROP { TABLE, EXTENSION, POLICY, ... }`
    - 36 DROP commands

How can you capture DDL?

- Use event triggers!
- ... which are a relatively new PostgreSQL feature
  - introduced in 9.3
- runs a user-defined function when a database event occurs

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 318633

Syntax of CREATE EVENT TRIGGER

```
CREATE EVENT TRIGGER name
  ON event
  [ WHEN filter_variable IN (filter_value [, ... ])
       [ AND ... ] ]
  EXECUTE PROCEDURE function_name()
```

What events can occur?

- `ddl_command_start`
- `ddl_command_end`
- `sql_drop`
- `table_rewrite` (9.5)
- more could be added in the future

What can you do in the function?

- Anything you can do in a function
- PL/pgSQL is going to be most common
- Some magic variables with context info: `TG_TAG` and `TG_EVENT`
- Also other languages
- Set-returning functions for additional data

Trivial (useless) example

```
CREATE FUNCTION snitch() RETURNS event_trigger
LANGUAGE plpgsql VOLATILE AS $$
  BEGIN
    RAISE NOTICE 'we got a % event', TG_TAG;
  END;
$$;
CREATE EVENT TRIGGER snitch
  ON ddl_command_end
EXECUTE PROCEDURE snitch();
```

Trivial (useless) example

```
CREATE FUNCTION snitch() RETURNS event_trigger
LANGUAGE plpgsql VOLATILE AS $$
  BEGIN
    RAISE NOTICE 'we got a % event', TG_TAG;
  END;
$$;
CREATE EVENT TRIGGER snitch
  ON ddl_command_end
EXECUTE PROCEDURE snitch();

alvherre=# CREATE TABLE mytable (col INTEGER);
NOTICE: we got a CREATE TABLE event
```

Trivial (useless) example

```
CREATE FUNCTION snitch() RETURNS event_trigger
LANGUAGE plpgsql VOLATILE AS $$
  BEGIN
    RAISE NOTICE 'we got a % event', TG_TAG;
  END;
$$;
CREATE EVENT TRIGGER snitch
  ON ddl_command_end
EXECUTE PROCEDURE snitch();

alvherre=# CREATE TABLE mytable (col INTEGER);
NOTICE: we got a CREATE TABLE event

alvherre=# create schema schema_b
           create table foo (a int)
           create table bar (b int);
NOTICE: we got a CREATE SCHEMA event
```

Dropped objs: `pg_event_trigger_dropped_objects()`

- Can be used in `sql_drop` only

column	*data type*
classid	oid
objid	oid
objsubid	oid
original	boolean
normal	boolean
is_temporary	boolean
object_type	text
schema_name	text
object_name	text
object_identity	text
address_names	text[]
address_args	text[]

Dropped Objects: Result Columns

classid, objid, objsubid — OID-based object identifier

original — whether the object was direct target of the DROP

normal — whether there's a "normal" dependency path from the parent object to this one (for instance, in a DROP TABLE there's a path to each index)

is_temporary — whether the object was temporary

object_type — "table", "schema" etc

schema_name — containing schema of the object, or NULL

object_name — name of the object

object_identity — machine-readable object identity

address_names, address_args — can be passed to `pg_get_object_address` to retrieve OID-based object identifier

Dropped Objects: More

pg_identify_object(classid, objid, objsubid) obtain machine-readable string object identity

pg_get_object_address(object_type, address_names, address_args) obtain OID-based object identity

- Note the sql_drop event runs after deletion has occured
- the object is no longer in system catalogs
- Consider `DROP SCHEMA`
- Consider `DROP OWNED BY`

```
CREATE FUNCTION report_drop() RETURNS event_trigger
LANGUAGE plpgsql AS $$
DECLARE
  r RECORD;
BEGIN
  FOR r IN SELECT * FROM pg_event_trigger_dropped_objects()
  LOOP
    RAISE NOTICE 'dropped: type "%" identity %',
                 r.object_type, r.object_identity;
  END LOOP;
END;
$$;

CREATE EVENT TRIGGER report_drop
  ON sql_drop EXECUTE PROCEDURE report_drop();
```

```
alvherre=# ALTER TABLE mytable DROP COLUMN col;
NOTICE: dropped: type "table column"
                 identity public.mytable.col
NOTICE: we got a ALTER TABLE event
```

DDL commands: `pg_event_trigger_ddl_commands()`

column	*data type*
classid	oid
objid	oid
objsubid	oid
command_tag	text
object_type	text
schema_name	text
object_identity	text
in_extension	boolean
command	pg_ddl_command

DDL Commands: Columns

classid, objid, objsubid same as before

command_tag `CREATE FUNCTION`, etc

object_type `function`, etc

schema_name name of the containing schema, or NULL

object_identity machine-readable identity

in_extension whether the command executes in an extension script

command magic!

Sample event trigger code

```
CREATE FUNCTION snitch() RETURNS event_trigger
LANGUAGE plpgsql VOLATILE AS $$
  DECLARE
    r RECORD;
  BEGIN
    FOR r IN SELECT * FROM pg_event_trigger_ddl_commands() LOOP
      RAISE NOTICE 'we got a % event for object "%"',
        r.command_tag, r.object_identity;
    END LOOP;
  END;
$$;

CREATE EVENT TRIGGER snitch
  ON ddl_command_end
EXECUTE PROCEDURE snitch();
```

Example execution

```
$= CREATE TABLE tab (col1 INT);
NOTICE: we got a CREATE TABLE event for object "public.tab"
```

Example execution

```
$= CREATE TABLE tab (col1 INT);
NOTICE: we got a CREATE TABLE event for object "public.tab"

$= CREATE SCHEMA sch
         CREATE TABLE foo (a serial)
         CREATE TABLE bar (b integer);

NOTICE: we got a CREATE SCHEMA event for object "sch"
NOTICE: we got a CREATE SEQUENCE event for object "sch.foo_a_seq"
NOTICE: we got a CREATE TABLE event for object "sch.foo"
NOTICE: we got a ALTER SEQUENCE event for object "sch.foo_a_seq"
NOTICE: we got a CREATE TABLE event for object "sch.bar"
CREATE SCHEMA
```

What is that `command` column again?

What is that `command` column again?

What is that `command` column again?

- column “command“ of type pg_ddl_command
- internal type, cannot be output directly
- pointer to a C struct
- can be passed to a C-language function for further processing
- `https://www.postgresql.org/message-id/20150409161419.GC4369@alvh.no-ip.org`

pg_ddl_command

```
typedef struct CollectedCommand
{
    CollectedCommandType type;
    bool in_extension;
    Node *parsetree;

    union
    {
        /* most commands */
        struct
        {
            ObjectAddress address;
            ObjectAddress secondaryObject;
        } simple;

        /* GRANT / REVOKE */
        struct
        {
            InternalGrant *istmt;
        } grant;

        /* ALTER TABLE */
        struct
        {
            Oid objectId;
            Oid classId;
            List *subcmds;
        } alterTable;
```

```
typedef enum CollectedCommandType
{
    SCT_Simple,
    SCT_AlterTable,
    SCT_Grant,
    SCT_AlterOpFamily,
    SCT_AlterDefaultPrivileges,
    SCT_CreateOpClass,
    SCT_AlterTSConfig
} CollectedCommandType;

typedef struct ObjectAddress
{
    Oid classId; /* Class Id from pg_class */
    Oid objectId; /* OID of the object */
    int32 objectSubId; /* Subitem within object */
} ObjectAddress;

typedef struct CollectedATSubcmd
{
    /* affected column, constraint, index, ... */
    ObjectAddress address;
    Node *parsetree;
} CollectedATSubcmd;
```

The JSON output

- We have an extension with a function that receives `pg_ddl_command` and returns JSON
- In the event trigger function you can modify the JSON
- We have a function to convert JSON back to text

A JSON blob

```
{"fmt": "CREATE %{persistence}s TABLE %{identity}D
          %{if_not_exists}s (%{table_elements:, }s)
          %{inherits}s %{on_commit}s %{tablespace}s",
   "persistence": "UNLOGGED",
   "identity": {
      "objname": "t1",
      "schemaname": "public" },
   "if_not_exists": {
      "fmt": "IF NOT EXISTS",
      "present": false },
   "inherits": {
      "fmt": "INHERITS (%{parents:, }D)",
      "parents": null,
      "present": false },
   "on_commit": {
      "fmt": "ON COMMIT %{on_commit_value}s",
      "on_commit_value": null,
      "present": false },
   "table_kind": "plain",
   "tablespace": {
      "fmt": "TABLESPACE %{tablespace}I",
      "present": false,
      "tablespace": null }
}
```

A JSON blob (2)

```
"fmt": " ... (%{table_elements:, }s) ..."
"table_elements": [
    {
        "fmt": "%{name}I %{coltype}T %{default}s
                %{not_null}s %{collation}s",
        "type": "column"
        "name": "a",
        "not_null": "NOT NULL",
        "collation": {
            "fmt": "COLLATE %{name}D",
            "present": false
        },
        "coltype": {
            "is_array": false,
            "schemaname": "pg_catalog",
            "typename": "int4",
            "typmod": ""
        },
        "default": {
            "default": "nextval('t1_a_seq'::regclass)",
            "fmt": "DEFAULT %{default}s"
        },
    }
```

Possible “fmt” escapes

%%	expand to a literal
%{name}I	expand as a single, non-qualified identifier
%{name}D	expand as a possibly-qualified identifier
%{name}T	expand as a type name
%{name}O	expand as an operator name
%{name}L	expand as a string literal (quote using single quotes)
%{name}s	expand as a simple string (no quoting)
%{name}n	expand as a simple number (no quoting)
%{name}R	expand as a role name (possibly quoted name, or PUBLIC)

Helper functions

- `pg_event_trigger_expand_command(jsonb)`

```
CREATE UNLOGGED TABLE public.t1
     (a pg_catalog.int4
      DEFAULT nextval('t1_a_seq'::regclass)
      NOT NULL )
```

- `jsonb_set(json, path, value)`

The AXLE project

Advanced Analytics for eXtremely Large European Databases

`http://www.axleproject.eu/`

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 318633

2ndQuadrant
Professional PostgreSQL

Questions?

- Thanks for listening
- Please give feedback on the JSON side of things

this page intentionally left blank

Event Triggers: Development History

- *thread:* `[HACKERS] Command Triggers`
  135 messages, 10 patch versions: Nov 2011 – Mar 2012

  `http://www.postgresql.org/message-id/m2pqh2mrq6.fsf@2ndQuadrant.fr`
- *thread:* `[HACKERS] Command Triggers, patch v11`
  115 messages, 5 patch versions: Feb 2012 – Mar 2012

  `http://www.postgresql.org/message-id/m24nufq466.fsf@2ndQuadrant.fr`
- *thread:* `[HACKERS] Command Triggers, v16`
  51 messages, 1 patch version: March 2012

  `http://www.postgresql.org/message-id/m2r4wtohqn.fsf@2ndQuadrant.fr`
- `Subject: [HACKERS] Command Triggers (v17)`
  1 message, 1 patch version: March 2012

  `http://www.postgresql.org/message-id/m23995lerl.fsf@2ndQuadrant.fr`
- *thread:* `[HACKERS] Command Triggers patch v18`
  43 messages, 1 patch version: Mar 2012 – Apr 2012

  `http://www.postgresql.org/message-id/m2sjh35fj7.fsf@2ndQuadrant.fr`

Event Triggers: Development History

- *thread:* `[HACKERS] Event Triggers reduced, v1`
  64 messages, 10 patch versions: Jun 2012 – Aug 2012

  `http://www.postgresql.org/message-id/m2aa04jp6h.fsf@hi-media.com`
- *commit:* Syntax support and documentation for event triggers.
  Date: Wed Jul 18 10:16:16 2012 -0400

  `http://git.postgresql.org/pg/commitdiff/3855968f328918b6cd1401dd11d109d471a54d40`
- *commit:* Make new event trigger facility actually do something.
  Date: Fri Jul 20 11:38:47 2012 -0400

  `http://git.postgresql.org/pg/commitdiff/3a0e4d36ebd7f477822d5bae41ba121a40d22ccc`

Event Triggers: Development History

- *thread:* `[HACKERS] Event Triggers: adding information`
  114 messages, 9 patch versions: December 2012

  `http://www.postgresql.org/message-id/m2txrsdzxa.fsf@2ndQuadrant.fr`
- *commit:* Add ddl_command_end support for event triggers.
  Date: Mon Jan 21 18:00:24 2013 -0500

  `http://git.postgresql.org/pg/commitdiff/841a5150c575ccd89e4b03aec66eeeefb21f3cbe`

Event Triggers: Development History

- *thread:* `[HACKERS] sql_drop Event Trigger`
  94 messages, 13 patch versions: Jan 2013 – Mar 2013

  `http://www.postgresql.org/message-id/m2fw1ieq5x.fsf@2ndQuadrant.fr`
- *commit:* Allow extracting machine-readable object identity
  Date: Wed Mar 20 18:19:19 2013 -0300

  `http://git.postgresql.org/pg/commitdiff/f8348ea32ec8d713cd6e5d5e16f15edef22c4d03`
- *commit:* Add sql_drop event for event triggers
  Date: Thu Mar 28 13:05:48 2013 -0300

  `http://git.postgresql.org/pg/commitdiff/473ab40c8bb3fcb1a7645f6a7443a0424d70fbaf`

Event Triggers: Development History

- *thread:* `[HACKERS] Add CREATE support to event triggers`
  106 messages, 9 patch versions: Nov 2013 – Jun 2014

  `http://www.postgresql.org/message-id/20131108153322.GU5809@eldon.alvh.no-ip.org`
- *thread:* `[HACKERS] deparsing utility commands`
  54 messages, 12 patch versions: February 2015 – May 2015

  `http://www.postgresql.org/message-id/20150215044814.GL3391@alvh.no-ip.org`
- *commit:* Allow on-the-fly capture of DDL event details
  Date: Mon May 11 19:14:31 2015 -0300

  `http://git.postgresql.org/pg/commitdiff/b488c580aef4e05f39be5daaab6464da5b22a494`