SF POSTGRES

2ndQuadrant
Professional PostgreSQL

Logical Decoding and Auditing

Gianni Ciolli

FOSS4G North America 2015
PostgreSQL Theme Day
Burlingame, 10 March 2015

Feature History

- 7.0 and older
  - Changes written to 1+ files on commit
  - Random writes
  - Changes are not collected anywhere

- 7.1 (2001): **W**rite **A**head **L**og
  - All changes “serialized” into *one* sequence
  - Sequential writes to *WAL files*
  - Changes are collected in binary format

Feature History #2

- 8.0 (2005): **P**oint **I**n **T**ime **R**ecovery
  - WAL files copied to the *archive*
  - Replay changes on another database server
  - The whole database server is cloned

- 8.2 (2006): Warm Standby
  - While replaying changes, waits for next WAL file
  - The clone is continuously updated. . .
  - . . . a.k.a. Replication

Feature History #3

- 9.0 (2010): Hot Standby, Streaming Replication
  - While replaying changes, read-only access
  - Changes are streamed using a client connection
- 9.4 (2014): Logical Decoding
  - Changes are streamed in *logical* format

Binary Changes: WAL

- Example:

  «write `0010010010110100...` into file A at offset B»

- Very fast
  - Only which bytes have changed, and how
  - No SQL, very little logical information

- Not flexible
  - Each changes depends on the previous one
  - Changes must be applied to become meaningful
  - Changes cannot be modified safely
  - Cannot merge changes from different systems

Logical Changes

- Example:

  «Insert string ‘Hello’ into table T»

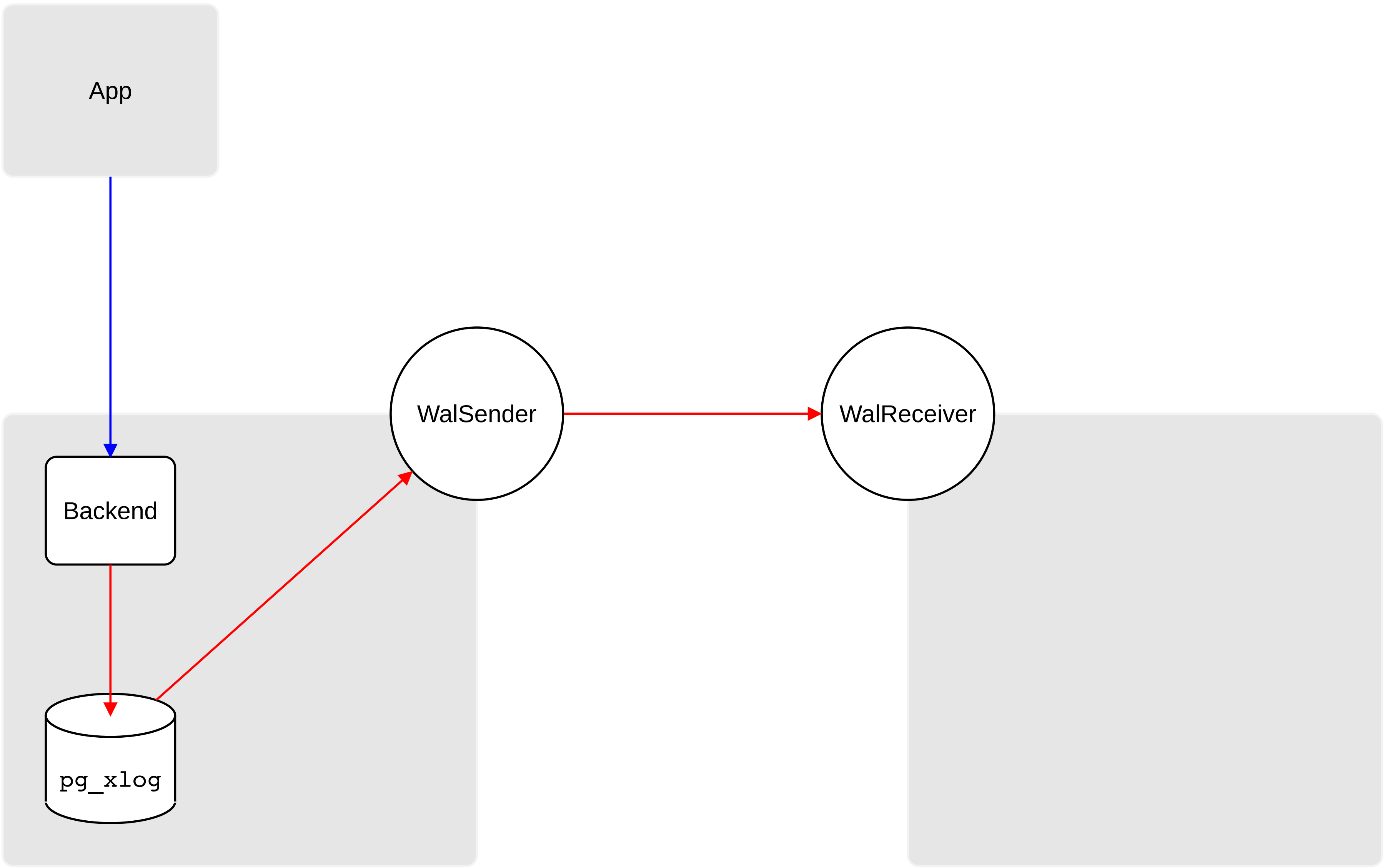
- Logical changes can now be *understood*

- Open up many possibilities:
  - Changes can be analysed
  - . . . can be modified
  - . . . can be reordered (with reason!)
  - . . . can be merged with other changes

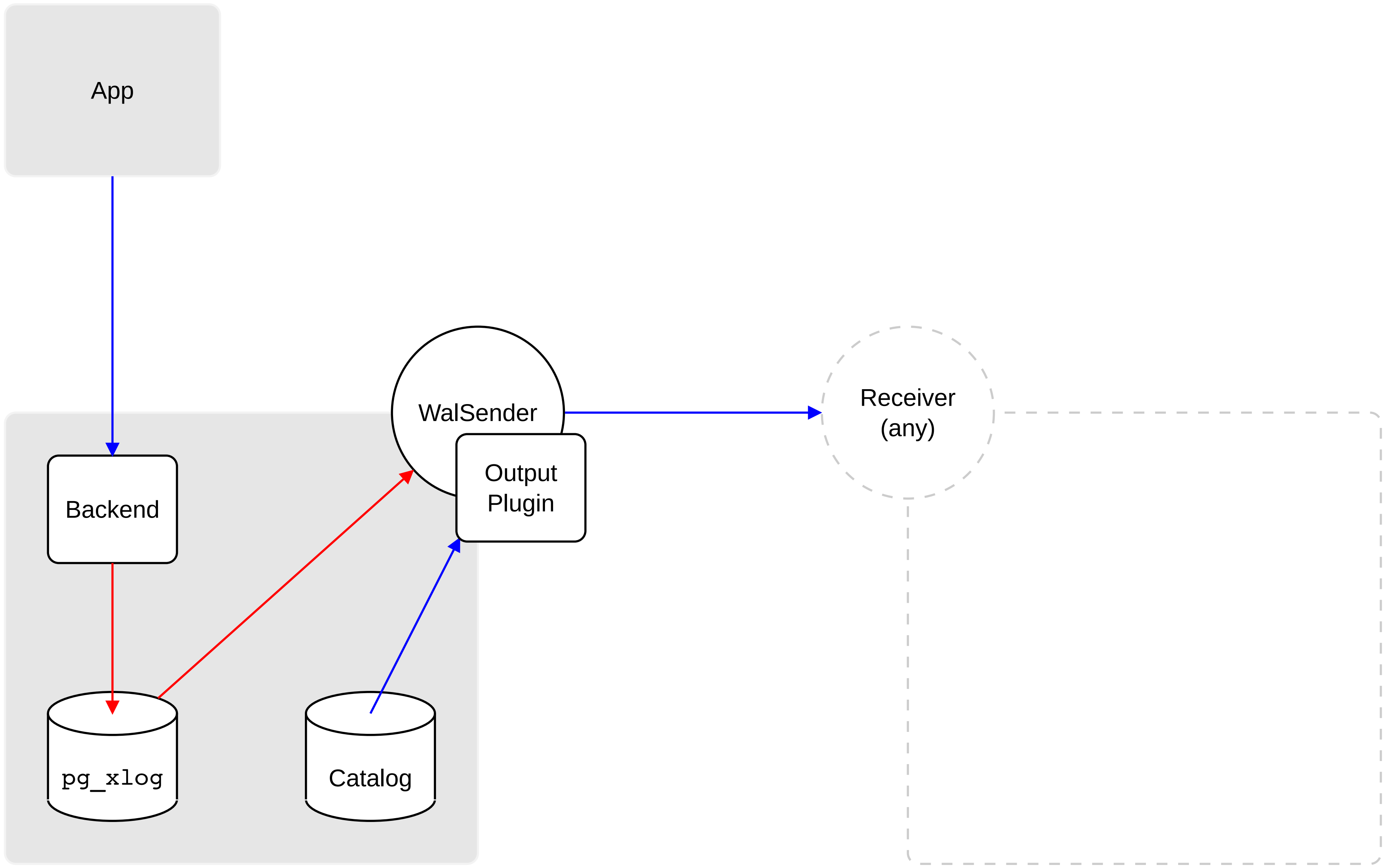
How Logical Works

- Decoding
  - WAL describes *file* changes
  - WAL is **decoded** to *table* changes
  - DML only: the rest is ignored!
- Tables ↔ Files
  - Mapping required for decoding
  - Defined in the *catalog*
- Output
  - Logical decoding **transforms** data
  - Changes are **streamed** by `walsender`

Binary Changes



Logical Decoding



Use Case: Replication

- **Selective**
  Filter by table and more
- **Bi-Directional**
  Conflict resolution now possible
  http://www.2ndQuadrant.com/BDR
- **Uni-Directional**
  Why? Less restrictions than Binary
  (Online upgrades, temp tables, . . .)

Other Use Cases

- “Logical Archiving”
- Diagnostics
- Auditing
  - Topic of the remaining slides!

Logical (v Binary)

Binary	Logical
instance	database
DML, DDL, . . .	DML only
only NEW	also OLD

- Uses the catalogue (hence database-wide)
- Capture (some) DDL with Event Triggers
- “Forwards” and “backwards”
  - For `UPDATE` and `DELETE`
  - `ALTER TABLE ... REPLICA IDENTITY` controls amount of OLD

Time Travel ???

- Could implement “time travel”. . .

WARNING

- PostgreSQL **had** time travel!
- Removed **17 years** ago
  - Performance reasons
  - There is **even** an extension. . .
- Before coding:
  - Evaluate costs v benefits
  - Check history. . .

REPLICA IDENTITY

```
ALTER TABLE myTable
  REPLICA IDENTITY ...;
```

- Which “old” column values?
  - NOTHING
    - None
  - FULL
    - All
  - USING INDEX myIndex
    - Columns covered by this index
  - DEFAULT
    - Primary key columns (if any)

Minimal Example #1 (config)

```
ALTER SYSTEM
  SET wal_level = logical;

ALTER SYSTEM
  SET max_replication_slots = 10;

-- Then restart...
```

Minimal Example #2 (SQL)

- Step 1: create a logical replication slot

```
SELECT * FROM
  pg_create_logical_replication_slot
    (’slot1’,’test_decoding’);


 slot_name | xlog_position
-----------+---------------
 slot1     | 0/1AE67D4
(1 row)
```

Minimal Example #2 (SQL)

- Step 2: peek changes (same db)

```
SELECT * FROM
  pg_logical_slot_peek_changes
    ('slot1', NULL, NULL);

 location  | xid  |           data
-----------+------+-------------------------
 0/1B137EC | 9980 | BEGIN 9980
 0/1B18FCC | 9980 | COMMIT 9980
 0/1B18FCC | 9981 | BEGIN 9981
 0/1B18FCC | 9981 | table public.don_juan:
           |      |   INSERT:
           |      |   country[text]:'Spain'
           |      |   count[integer]:1003
 0/1B1904C | 9981 | COMMIT 9981
(5 rows)
```

Minimal Example #2 (SQL)

- What was the SQL?

```
CREATE TABLE don_juan (
country text NOT NULL,
count   int  NOT NULL );

INSERT INTO don_juan
  VALUES (’Spain’, 1003);
```

Minimal Example #2 (SQL)

- Step 3: when finished, drop the slot

```
SELECT pg_drop_replication_slot('slot1');
```

Auditing with Logical Decoding

- Single-database audit
  - Not a limitation actually!
- Performance
  - Very efficient
  - Generic benchmarks (A. Freund, P. Jelinek)

Auditing w. Logical Decoding #2

- What is logged?
  - No DSL
    - Difficult to audit anyway…
  - No DDL
    - Use Event Triggers for DDL
    - This is what BDR does
  - No DCL
    - Use Event Triggers for (some) DCL

Auditing w. Logical Decoding #3

- What is logged?
  - Not even DML…
  - … only the *consequences* of DML !
    - “row-based” view, not “statement-based”
    - no trace of `UPDATE` or `DELETE` hitting 0 rows
- Different solutions offer more coverage:
  - Event Triggers
  - The `pgaudit` extension

    https://github.com/2ndQuadrant/pgaudit

Auditing Mode I

- Via SQL interface
- Log to log tables
  - Do not log the logs!
  - From the same DB
  - **(!)** superuser can retrospectively alter logs
    - Really a downside???
- No separate service
  - Always up, cheaper to manage
  - Query and monitor in **real time**

Auditing Mode II

- Via external plugin
- Log outside
  - Logs cannot be retrospectively altered
  - “eventually superuser-safe”
- Separate service
  - Could be down
  - Must be managed
- Custom plugin, to avoid parsing text
  - Can be done in YourSetup v2.0

Output Plugin Example

- Start from the example in `contrib`:

```
/*-------------------------------------------------------------------------
 *
 * test_decoding.c
 *                example logical decoding output plugin
 *
 * Copyright (c) 2012-2014, PostgreSQL Global Development Group
 *
 * IDENTIFICATION
 *                contrib/test_decoding/test_decoding.c
 *
 *-------------------------------------------------------------------------
 */
#include "postgres.h"

#include "access/sysattr.h"

#include "catalog/pg_class.h"
#include "catalog/pg_type.h"

...
```

And now. . .

Questions?

And then. . .

Thank you!

Feedback here:

`http://2015.foss4g-na.org/`

Licence