

Postgres Replication

Andres Freund

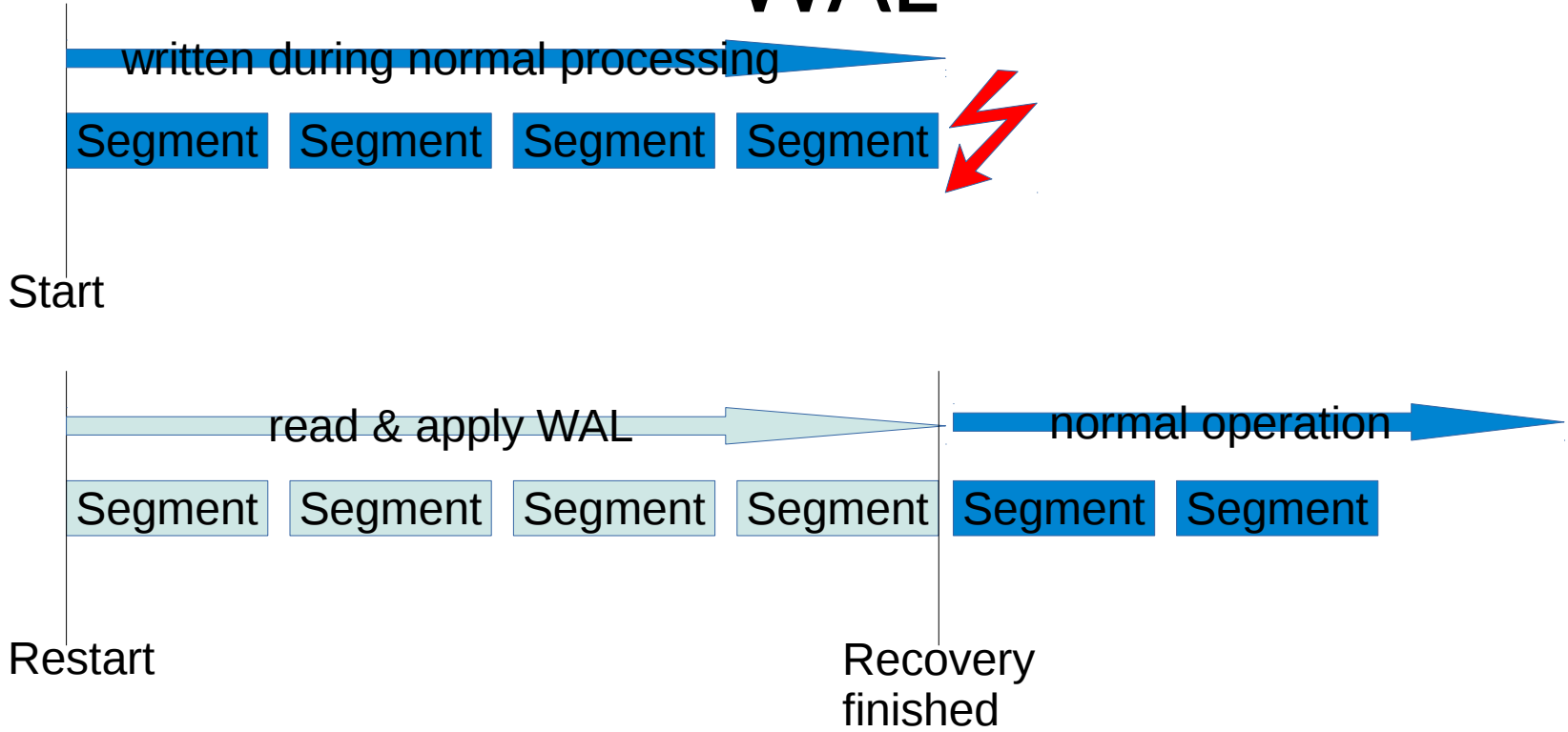
PostgreSQL Contributor & Committer

Working at Citus Data

Streaming Replication Basics

- Based on Crash Recovery Mechanism
- Writes are only possible on primary
- Whole “cluster”
- WAL has to be available
- Start from a basebackup

WAL



Prerequisite Configuration

- `max_wal_senders = 10`
- `wal_level = 'hot_standby'`
- **Replication access in `pg_hba.conf`**
`<type> <db> <user> <src> <meth>`
`host replication andres 127.0.0.1/32 md5`

Creating a clone

- `pg_basebackup` - by far the easiest
- `pg_start_backup()` ;
<copy data, eg. using `rsync`>
`pg_stop_backup()` ;

Copy node using pg_basebackup

```
~$ pg_basebackup \  
  -d 'host=localhost port=6001 user=andres' \  
  -D /tmp/test/standby-1 \  
  -X stream \  
  --checkpoint=fast \  
  --progress
```

- `-d` - database/connection string
- `-D` - destination direction
- `-X stream` - copy required WAL
- `--checkpoint=fast | spread` - impact on primary

Basebackup – now what?

- new basebackup is independent
- configure to make it a standby
- most replication settings are configured in `recovery.conf`
- `recovery.conf` triggers “recovery mode”

Configure basebackup as replica

- `postgresql.conf: hot_standby = on`
- `recovery.conf:`
`standby_mode = 1`
`primary_conninfo = 'host=localhost`
`port=6001 user=andres'`

The dreaded FATAL

- FATAL: could not receive data from WAL stream:
ERROR: requested WAL segment 0000000100000000000000031 has already been removed
- WAL has to be available to be streamed!
- Three methods to preserve WAL
 - wal_keep_segments – fixed number of WAL
 - archive_command/restore_command
 - replication_slots

wal_keep_segments

- fixed number of 16MB chunks preserved
- trivial to set, hard to know what to set to
- must be overestimated

Archiving

- Copy files to standby/other location
- restore files during recovery
- `archive_command` - copy data from the master to standby/somewhere (`postgresql.conf`)
- `restore_command` - copy data from somewhere to target location on the standby (`recovery.conf`)
- Complex and powerful
- `restore_command` can be slow over network, bursty access

Replication Slots

- New in 9.4

- **primary postgresql.conf:**

```
max_replication_slots = 10
```

```
hot_standby_feedback = on
```

- **Create slot for standby on primary**

```
SELECT pg_create_physical_replication_slot(  
        'andrescorp_db_standby1');
```

- **standby recovery.conf:**

```
primary_slot_name = 'andrescorp_db_standby1'
```

Replication Slot

- A slot is a resource reservation
- Reserves all required WAL
- One standby per slot
- Keeps WAL forever if standby is gone unless removed!

Failover

- Promote standby:

```
pg_ctl -D ... promote
```

- Cleanup slot on master

```
SELECT pg_drop_replication_slot(...)
```

- Failing back/repointing old master is hard < 9.5
- pg_rewind can fail back in 9.5

Streaming Rep Pros

- Builtin
- FAST
- low overhead
- Once correctly setup pretty simple
- Robust due to reused crash recovery mechanism
- Schema is always correctly replicated
- Mature – Many, many deployments in the field

Streaming Rep Cons

- All databases, all tables, everything is replicated
- No temp tables or anything possible on the standby
- No cross version/architecture replication
- Reads on the standby depend on locks on the standby

Logical Decoding

- AKA changeset extraction; change data capture
- All changes to a database in commit order
- Configurable output format
- Low level API
- Efficient
- More features than possible with pqg etc.

What are UDR/BDR

- Logical replication solutions
- Built on top of logical decoding
- We move as much as possible into core PG Database, or smaller, scope
- Created by 2ndQuadrant; with some external contributions
- Funding, amongst others, from Intel Security, Adyen

How do things fit together

BDR:

- Active/Active
- DDL Replication
- Conflict resolution
- Distributed Sequences

Modified
PG 9.4+

UDR:

- Primary/Standby
- Efficient
- Cross Version 9.4+
- Init from physical clone

Stock PG 9.4+
Extension

Postgresql 9.4:

- Logical Decoding
- Background Workers

Using UDR

- 1) Basic configuration on all nodes
- 2) Subscribe standby to primary
- 3) Subscribing clones all data, while writes are continuing to go on on the master
- 4) Cloning can be logical or physical

Prerequisite Configuration

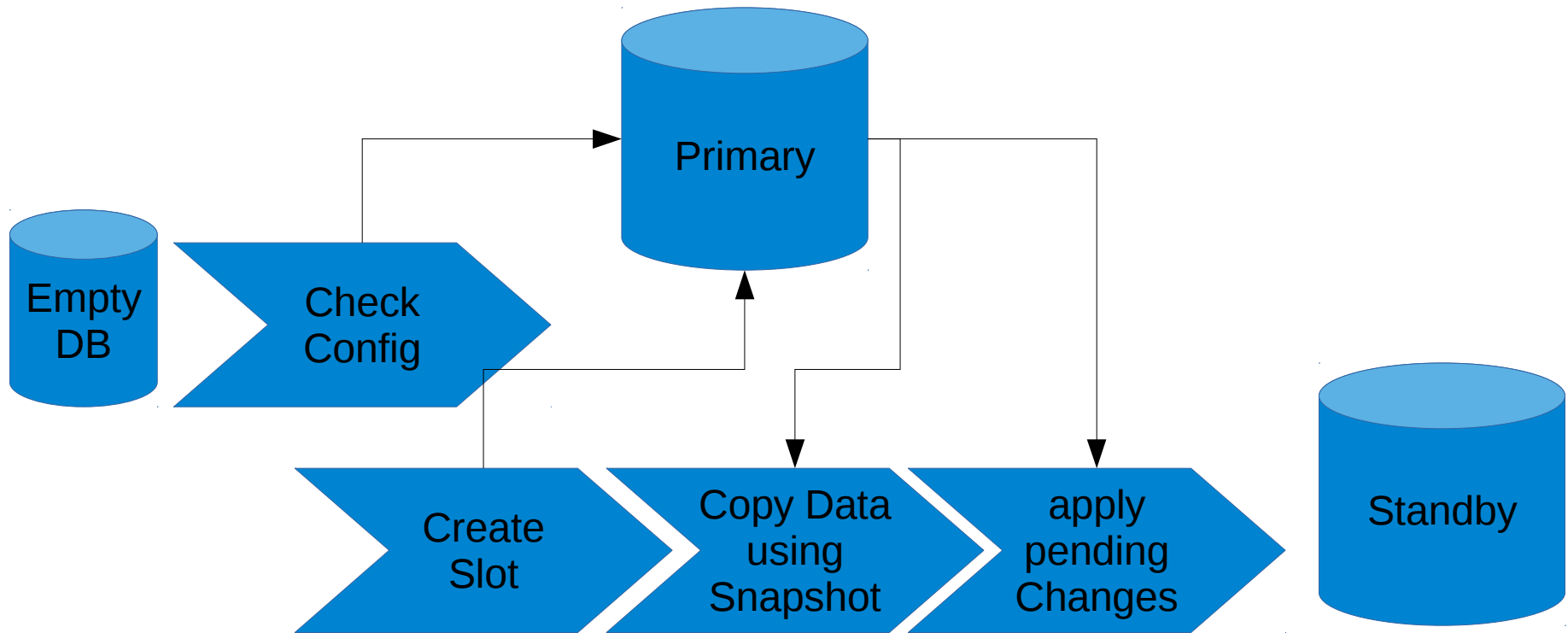
- `max_wal_senders = 10`
- `max_replication_slots = 10`
- `shared_preload_libraries = 'bdr'`
- `wal_level = 'logical'`
- **Replication access in `pg_hba.conf`**

Preparing the Primary

```
Postgres=# SHOW wal_level;  
postgres=# SHOW max_replication_slots;  
postgres=# CREATE EXTENSION btree_gist;  
postgres=# CREATE EXTENSION bdr;  
postgres=# CREATE TABLE testtable(key text primary key);  
postgres=# INSERT INTO testtable(key) VALUES ('a');
```

Creating the Standby

```
postgres=# CREATE EXTENSION btree_gist;
postgres=# CREATE EXTENSION bdr;
postgres=# SELECT bdr.bdr_subscribe(
    local_node_name := 'node-standby-01',
    subscribe_to_dsn := 'host=primary',
    node_local_dsn := 'host=standby-01'
);
postgres=# bdr.bdr_node_join_wait_for_ready();
Postgres=# \d testtable
```



Executing DDL

```
postgres=# CREATE TABLE fail();  
postgres=# SELECT bdr.bdr_replicate_ddl_command($DDL$  
        CREATE TABLE public.nofail();  
$DDL$);
```

Uncouple/Promote Standby

```
postgres=# SELECT bdr.bdr_part_by_node_names (  
    ARRAY [ 'node-standby-01' ]  
);
```

Replication Sets – The Why

- Often only part of the data is important
 - Session tables, long lived temp tables, log data
- Different destinations want different data
 - Local Availability: everything
 - Failover to other DC: important data

Replication Sets – What & How

- Every table can be in multiple sets
- Every subscriber can subscribe to one or more sets
- The 'default' set contains all tables without explicit configuration
- The 'all' set ...

Replicating Sets

```
postgres=# SELECT bdr.bdr_subscribe(  
  local_node_name := 'node-standby-01',  
  subscribe_to_dsn := 'host=primary',  
  node_local_dsn := 'host=standby-01',  
  replication_sets := ARRAY['default', 'important']  
);
```

Configuring Tables

```
postgres=# CREATE TABLE users(...);
postgres=# CREATE TABLE logins(...);
postgres=# CREATE TABLE sessiondata(... );
/* cloned here */
postgres=# bdr.table_set_replication_sets('
  'sessiondata', ARRAY['volatile']);
postgres=# bdr.table_set_replication_sets('
  'users', ARRAY['important']);
postgres=# bdr.table_set_replication_sets('
  'logins', ARRAY['logdata', 'volatile']);
```

Replication Sets – Caveat!

- All tables are cloned initially
- Only subscribed to tables are updated
- Changing set subscription requires a restart
- No builtin re-synchronization after set changes

UDR - Pros

- Fairly flexible
- Low overhead
- Cross Version Replication

UDR Cons

- Quite new < 1.0
- Batch loads can delay replication of other xacts

BDR

- Very similar!
- No master, no subscription to a single node
- First node: create group
- Further nodes: join group

Prerequisite Configuration

- `max_wal_senders = 10`
- `max_replication_slots = 10`
- `shared_preload_libraries = 'bdr'`
- `wal_level = 'logical'`
- **Replication access in `pg_hba.conf`**
- `track_commit_timestamps = on`
- `bdr.log_conflicts_to_table = on`

Create BDR Group:

```
postgres=# CREATE EXTENSION btree_gist;
postgres=# CREATE EXTENSION bdr;
postgres=# SELECT bdr.bdr_group_create(
    local_node_name := 'node-01',
    node_external_dsn := 'host=node-01');
```

Join BDR Group:

```
postgres=# CREATE EXTENSION btree_gist;
postgres=# CREATE EXTENSION bdr;
postgres=# SELECT bdr.bdr_group_join(
    local_node_name := 'node-02',
    node_external_dsn := 'host=node-02',
    join_using_dsn := 'host=node-01',
);
\c another host
postgres=# SELECT bdr.bdr_group_join(
    local_node_name := 'node-03',
    node_external_dsn := 'host=node-03',
    join_using_dsn := 'host=node-01',
);
```

Executing DDL

```
postgres=# CREATE TABLE nofail();  
postgres=# ALTER TABLE nofail ADD COLUMN bar text;
```

DDL Replication

- Most database local objects can be replicated
- If not: ERROR
- Global schema lock
- Cluster wide objects can't be replicated!
- Important infrastructure in 9.5! UDR will be able to do it!

Conflicts

- Asynchronous Multi-Master => Conflicts
- Last update wins using unambiguous timestamps
- Conflict triggers/handlers
- Conflicts can be logged to log table

Distributed Sequences

- `CREATE/ALTER SEQUENCE ... USING bdr;`
- Chunked sequence allocations
 - Node 01: 1-1000, 3001-4000
 - Node 02: 2001-3000
- Chunks are pre-allocated
- Integrate abstraction into 9.6?

Monitoring

- `bdr.pg_stat_bdr`
- `bdr.bdr_node_slots`
- `pg_replication_slots`
- `pg_stat_replication`
- `bdr.bdr_conflict_history`

Physical Copy

- `bdr_init_copy`
- Uses `pg_basebackup` or existing basebackup
- Only one database setup for replication
- Works while writes are in progress:
 - create base backup
 - create logical slots
 - physical catchup
 - logical catchup

Cross Version Upgrades

- Rolling upgrades, yay!
- Replication protocol version independent
- Fallback to less efficient transport across versions
- DDL replication is harder across versions
- Works with logical clone
- Some work remains for physical clones

Where is BDR/UDR going?

- More granular clones/recloning tables upon set changes
- Doc improvements
- Usability improvements
- DDL replication for UDR 9.5
- “2PC Multi-Master” => optional conflict free apply
- Propose minimal version, infrastructure, for integration

UDR/BDR Resources

- Docs: bdr-project.org/docs
- Product info: bdr-project.org
- Support available by 2ndQ
- Bugs: github.com/2ndQuadrant/bdr/issues/