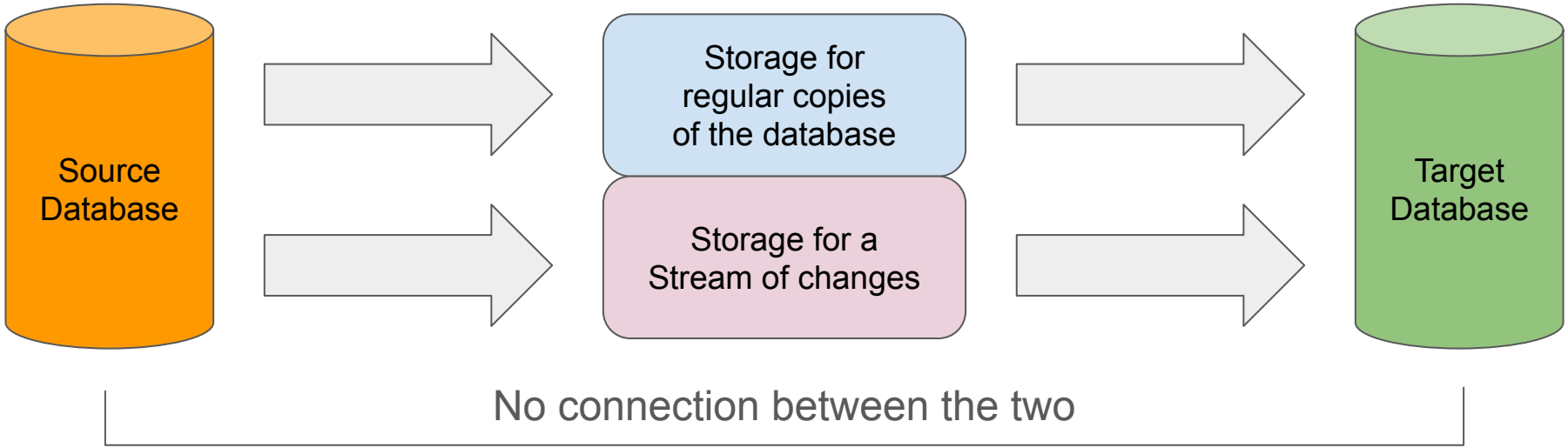
OwnYourPG

A journey in logical replication

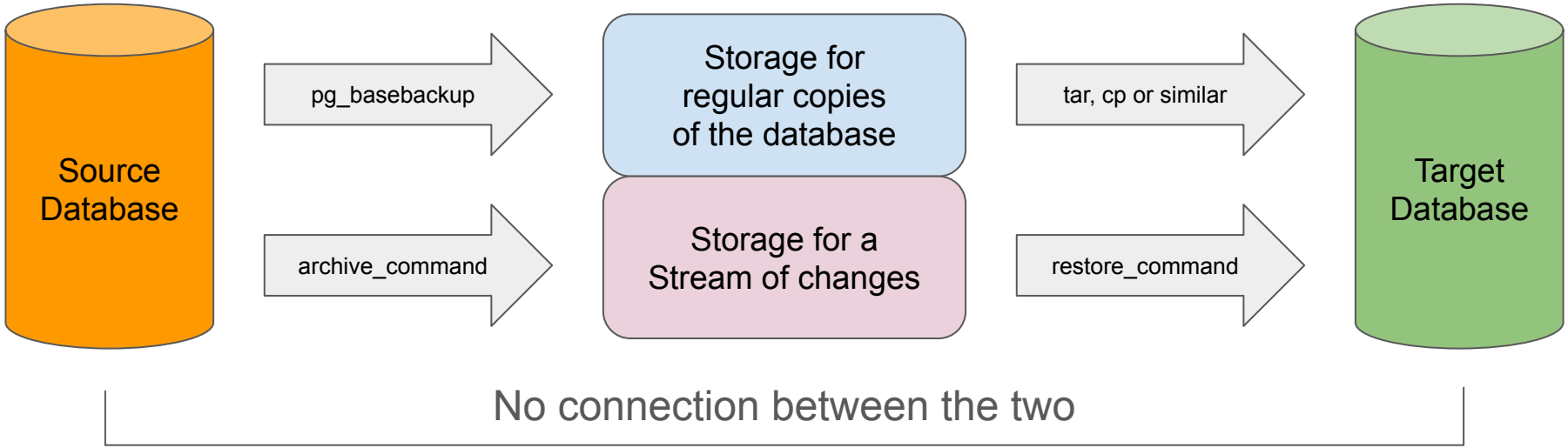
Boredom and a problem

- PGConf.EU 2024 – Athens
- Move to Aurora

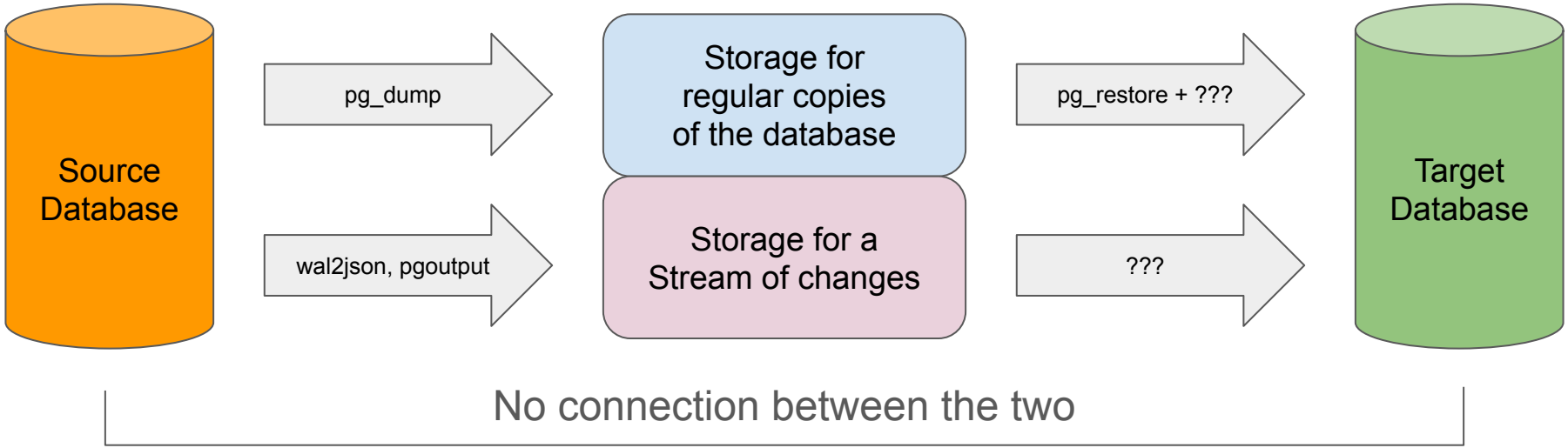
Point-in-time recovery



PITR – the classical picture



Logical PITR



Difference between physical and logical backup

Physical Backup (pg_basebackup)	**Logical Backup** (pg_dump)

`# some preparation`
 	
`cd $DATADIR &&`
`tar -cf - . >backup.tar`
 	
`# some cleanup`
	`CREATE TABLE my.table (…);`
	`COPY my.table FROM STDIN;`
	`Data row1`
	`Data row2`
	`\.`
	`CREATE INDEX … ON my.table(…);`

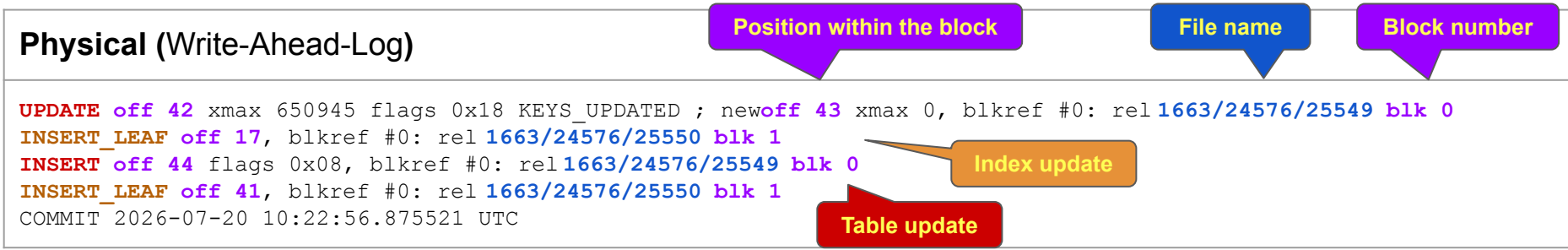
• Bit by bit copy • Needs the same environment to be restored (operating system, software versions, etc) • Per cluster • Contains data + metadata + indexes and similar • Can restore inconsistent data if the source is inconsistent	• SQL-level copy • Can be restored to a different operating system, across software versions. • Fine grained: per database, per table • Contains the structure and the data • Indexes, PK and FK constraints are rebuilt during restoration. • Consistency is checked during restoration

Physical vs. logical stream of changes

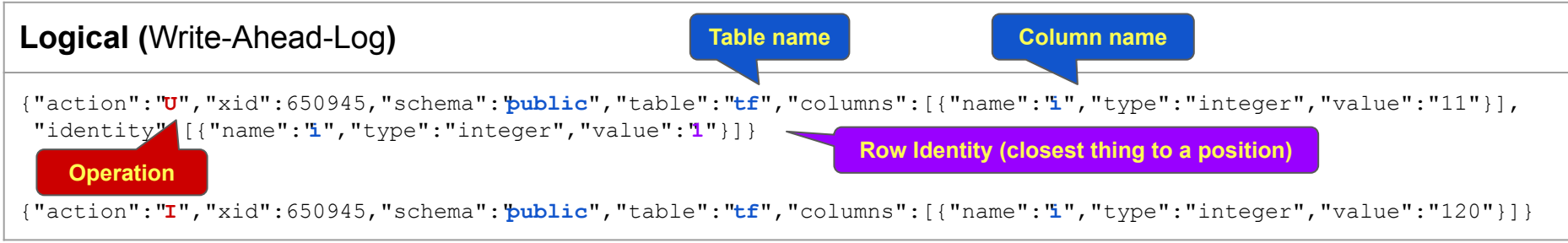
Operation

```
begin; update tf set i=i*11 where i=1; insert into tf(i) values (120); commit;
```

Physical (Write-Ahead-Log)



Logical (Write-Ahead-Log)



2 Problems

Initial Copy

- Copy a state of the database tied to a specific position in the stream of changes.

Capture and replay

- The mechanics of extracting the stream of changes and replaying it in a target database.

Classical PITR

- Unpack the base backup
- The DB knows its current WAL insert LSN
- Replay WAL segments just like in crash recovery
- The LSN is advanced automatically

Capture and replay

Wal2Json basics

Create a slot
```
select pg_create_logical_replication_slot('tmp', 'wal2json');
```

Some action
```
insert into tf select * from generate_series(2,4);
```

Peek

```
select data
  from pg_logical_slot_peek_changes(
          'tmp', null, null
       );
```

```
{
  "change":[
    {
      "kind":"insert",
      "schema":"public",
      "table":"tf",
      "columnnames":["i"],
      "columntypes":["integer"],
      "columnvalues":[2]
    },
    …
  ]
}
```

Wal2json on GitHub

Wal2json on GitHub

pg_recvlogical

```
pg_recvlogical -d postgres:///bench\?service=mydb -S tmp  --start -f- \
               -o format-version=2 -o include-lsn=true -o include-timestamp=true \
               -o include-xids=true -o numeric-data-types-as-string=true
```

```
insert into tf select * from generate_series(2,4);
```

```
{"action":"B","xid":650928,"timestamp":"2026-07-19 09:19:08.815035+00","lsn":"4/2A158E8",
 "nextlsn":"4/2A15918"}
{"action":"I","xid":650928,"timestamp":"2026-07-19 09:19:08.815035+00","lsn":"4/2A15220",
 "schema":"public","table":"tf","columns":[{"name":"i","type":"integer","value":2}]}
{"action":"I","xid":650928,"timestamp":"2026-07-19 09:19:08.815035+00","lsn":"4/2A157E8",
 "schema":"public","table":"tf","columns":[{"name":"i","type":"integer","value":3}]}
{"action":"I","xid":650928,"timestamp":"2026-07-19 09:19:08.815035+00","lsn":"4/2A15868",
 "schema":"public","table":"tf","columns":[{"name":"i","type":"integer","value":4}]}
{"action":"C","xid":650928,"timestamp":"2026-07-19 09:19:08.815035+00","lsn":"4/2A158E8",
 "nextlsn":"4/2A15918"}
```

Back to SQL

- 35 lines of jq code
- Including accounting
- Including minimal error handling

Gist on Github

```
read -rd '' JQ <<'EOF'
def dup(c): c as $c | gsub($c; 2*$c);
def mkname: "\"\(. | dup("\""))\"";
def mkvalue:
    if (. | type)=="boolean" then .|tostring elif (. | type)=="null" then .|tostring
    elif (. | type)=="number" then .|tostring else "'\(. | dup("'"))'" end;
def tbl: "\(.schema|mkname).\(.table|mkname)";
def cond:
     "(" + ([.identity[] | .name | mkname] | join(", ")) + ")" +
     " IS NOT DISTINCT FROM " +
     "(" + ([.identity[] | .value | mkvalue] | join(", ")) + ")";
def upd: [.columns[] | "\(.name|mkname) = \(.value|mkvalue)"] | join(", ");
def cols: [.columns[] | .name | mkname] | join(", ");
def vals: [.columns[] | .value | mkvalue] | join(", ");
def B: "BEGIN; SET LOCAL session_replication_role=replica;";
def C:
    "WITH upd AS ("+
    "UPDATE \($logtbl) SET lsn='\(.lsn)', xid='\(.xid)', stamp='\(.timestamp)'" +
    " WHERE origin='\($origin|dup("'"))' AND lsn < '\(.lsn|dup("'"))'" +
    " RETURNING *) SELECT 1/(count(*)=1)::INT FROM upd;\n" +
    "COMMIT;\n";
def I: "INSERT INTO \(.|tbl) (\(.|cols)) VALUES (\(.|vals));";
def U:
    "WITH cand AS (SELECT ctid FROM \(.|tbl) WHERE \(.|cond) FOR UPDATE LIMIT 1), "
+
    "upd AS (" +
    "UPDATE \(.|tbl) AS dest SET \(.|upd) FROM cand WHERE dest.ctid=cand.ctid" +
    " RETURNING *) SELECT 1/(count(*)=1)::INT FROM upd;";
def D:
    "WITH cand AS (SELECT ctid FROM \(.|tbl) WHERE \(.|cond) FOR UPDATE LIMIT 1), "
+
    "del AS (" +
    "DELETE FROM \(.|tbl) AS dest USING cand WHERE dest.ctid=cand.ctid" +
    " RETURNING *) SELECT 1/(count(*)=1)::INT FROM del;";
def T: "TRUNCATE \(.|tbl);";
if   .action=="B" then . | B elif .action=="C" then . | C elif .action=="I" then . |
I
elif .action=="U" then . | U elif .action=="D" then . | D elif .action=="T" then . |
T
elif .action=="M" then empty else . end
EOF
```

UPDATE

- `IS NOT DISTINCT FROM` handles NULL nicely
- `LIMIT 1` finds at most 1 row
- `FOR UPDATE` locks that row
- A ctid scan quickly finds the locked row in the UPDATE.
- `1/(count(*)=1)::INT` becomes a division by zero if more or less than one row was modified.

```
WITH cand AS (
    SELECT ctid
      FROM "public"."tf"
     WHERE ("i") IS NOT DISTINCT FROM ('5')
       FOR UPDATE
     LIMIT 1
),
upd AS (
    UPDATE "public"."tf" AS dest
       SET "i" = '4'
      FROM cand
     WHERE dest.ctid=cand.ctid
 RETURNING *
)
SELECT 1/(count(*)=1)::INT FROM upd;
```

COMMIT

- `w2j.log` keeps track of the position in the stream of changes.
- `1/(count(*)=1)::INT` in combination with the lsn check prevents repeating a transaction.

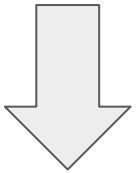
```
WITH upd AS (
    UPDATE w2j.log
       SET lsn='4/2A179F8'
         , xid='650935'
         , stamp='2026-07-19 13:18:41.398301+00'
     WHERE origin='w2j'
       AND lsn < '4/2A179F8'
 RETURNING *
)
SELECT 1/(count(*)=1)::INT FROM upd;
COMMIT;
```

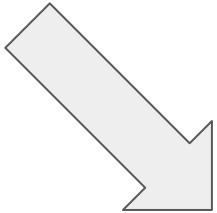
Initial Copy

Anchor a pg_dump in the WAL stream

Given a pg_dump and the JSON files, when do we need to start replaying?

- WAL = **ordered** stream of changes
- COMMIT records define visibility





Method 1

⇒ Tie it to the LSN of the COMMIT record of the most recently committed transaction

Method 2

⇒ Tie it to the LSN of the first not yet committed transaction

⇒ Same as anchoring it to an MVCC snapshot

Anchor a pg_dump to the most recently committed TXN

1. Create a physical replica (`pg_basebackup` or similar)
2. Cut replication (isolate the DB) but do not promote
3. `select pg_last_wal_replay_lsn()` and save the result
4. Run `pg_dump` against the replica

⇒ Transactions with COMMIT records with an LSN greater than the one saved in step 3 need to be replayed

What is an MVCC snapshot?

- `xmin` – the oldest (by creation time) not yet committed transaction
- `xmax` – the smallest not yet created transaction
- `xip_list` – a list of transaction ids between xmin and xmax that are still in progress (not visible from the point of view of the snapshot)

Example:

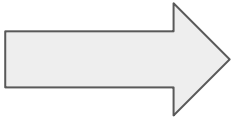
```
bench=*# select pg_current_snapshot();
pg_current_snapshot
----------------------
650936:650943:650936,650938,650939
(1 row)
```

- ➢ All transactions up to 650935 are committed or aborted. (`xmin-1`)
- ➢ Transactions 650943 and above are not yet born. (`xmax`)
- ➢ Transactions 650936, 650939 and 650938 are still in progress. (`xip_list`)
- ➢ Transactions not listed in `xip_list` are committed or aborted.

Exporting and importing a snapshot

- Exporting

```
bench*# select pg_export_snapshot();
pg_export_snapshot
---------------------
00000009-0000592D-1
(1 row)
```



```
$ cat pg_snapshots/00000009-0000592D-1
vxid:9/22829
pid:4024334
dbid:24576
iso:2
ro:0
xmin:650936
xmax:650943
xcnt:3
xip:650936
xip:650938
xip:650939
sof:0
sxcnt:0
rec:0
```

- Importing into pg_dump

```
$ pg_dump … --snapshot=00000009-0000592D-1
```

Linking a snapshot to an LSN

Replication connection (walsender)

```
$ psql 'postgres:///bench?service=mydb&replication=database'

bench=# CREATE_REPLICATION_SLOT tmp___ TEMPORARY LOGICAL pgoutput;
 slot_name | consistent_point |    snapshot_name    | output_plugin
-----------+------------------+---------------------+---------------
 tmp___    | 4/2A1CB58        | 00000009-00005992-1 | pgoutput
(1 row)
```

Snapshot for pg_dump

```
bench=# select … from pg_replication_slots where slot_name
-[ RECORD 1 ]-------+----------
database            | bench
temporary           | t
restart_lsn         | 4/2A1CA30
confirmed_flush_lsn | 4/2A1CB58
```

Replay starts here

```
 $ pg_dump … --snapshot=00000009-00005992-1
```

Expensive!

Simpler: start replay initially from the snapshot

- Create **REPEATABLE READ** transaction
- Export **snapshot** to **pg_dump**
- Keep the **transaction open** until **pg_dump** is done

- Pass the snapshot to the replay engine
- Any transactions `<xmin` does NOT need to be replayed
- Any transaction in `xip_list` or any transaction `>=xmax` starts the replaying

So far…

- pg_recvlogical extracts the stream of changes in JSON
- Able to convert JSON back to SQL – connection to primary not needed, only JSON files
- We know how to anchor a pg_dump at a WAL position

A few shell scripts later

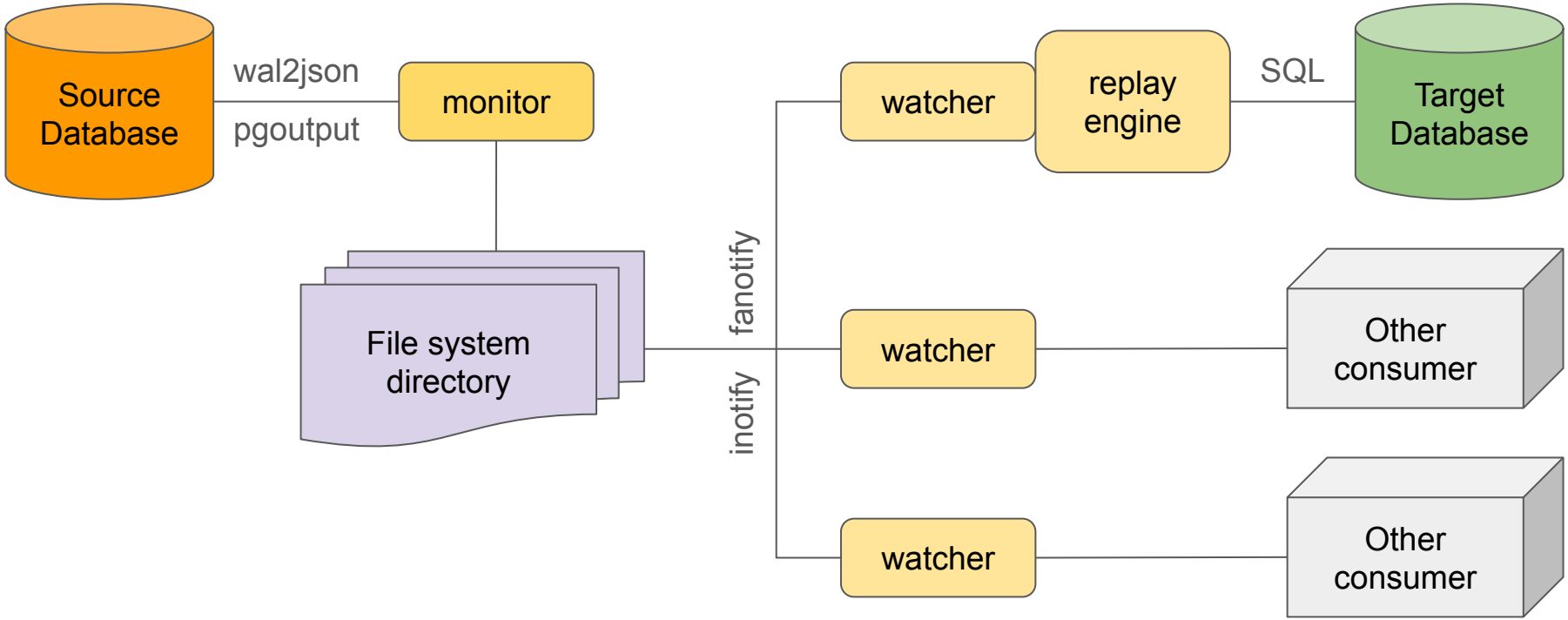
- Incredibly slow
- Single-threaded
- One statement per row

Quick optimization

- Combine consecutive INSERTs in COPY
- Skip COMMIT to build larger transactions

The journey

Where to?


Source Database
wal2json
pgoutput
monitor
File system directory
fanotify
inotify
watcher
replay engine
SQL
Target Database
watcher
Other consumer
watcher
Other consumer

Pg_recvlogical or ???

- SIGHUP to reopen the file
- Shell script:
  - Monitoring the file size
  - Renaming at specific size
  - Sending SIGHUP to reopen
- Too complicated
- Implementation in Go but with the same general idea
- Too little control when to advance the replication slot

⇒ Not reliable enough

- Rewrite in Go using `pgx + pglogrepl`
- Full control over `write_lsn` and `flush_lsn`
- File breaks strictly at a transaction boundary
- Better size control

SQL speed optimization

- `session_replication_role=replica`
- Pipeline mode eliminates network delay
- Join transactions
- Optimize transactions:
  - In chunks but within a transaction
  - UPDATE ⇒ DELETE + INSERT
  - Eliminate superfluous operations
  - Combine operations: 1 multi-row DELETE + 1 multi-row INSERT per table per chunk

Pgbench results

- 800-900 TXN/s on an unoptimized DB
- Replication delay <100 ms

Fingerprinting

- Select a subset of a table
- Calculate some sort of checksum
- Insert that checksum together with the query into the WAL stream
- Decode the WAL message on the replica
- Rerun the query
- Compare the checksums

```
v_acc := '';
FOR v_row IN
    EXECUTE $$
        SELECT t::TEXT
          FROM ($$ || query || $$) AS t
         ORDER BY t::TEXT COLLATE "POSIX"
    $$
LOOP
    v_acc := md5(v_acc || v_row);
END LOOP;
```

```
SELECT query_fingerprint($$
    SELECT * FROM table ORDER BY last_modified >= 'yesterday'
$$)
```

Gist on GitHub

How to insert the fingerprint message into the WAL

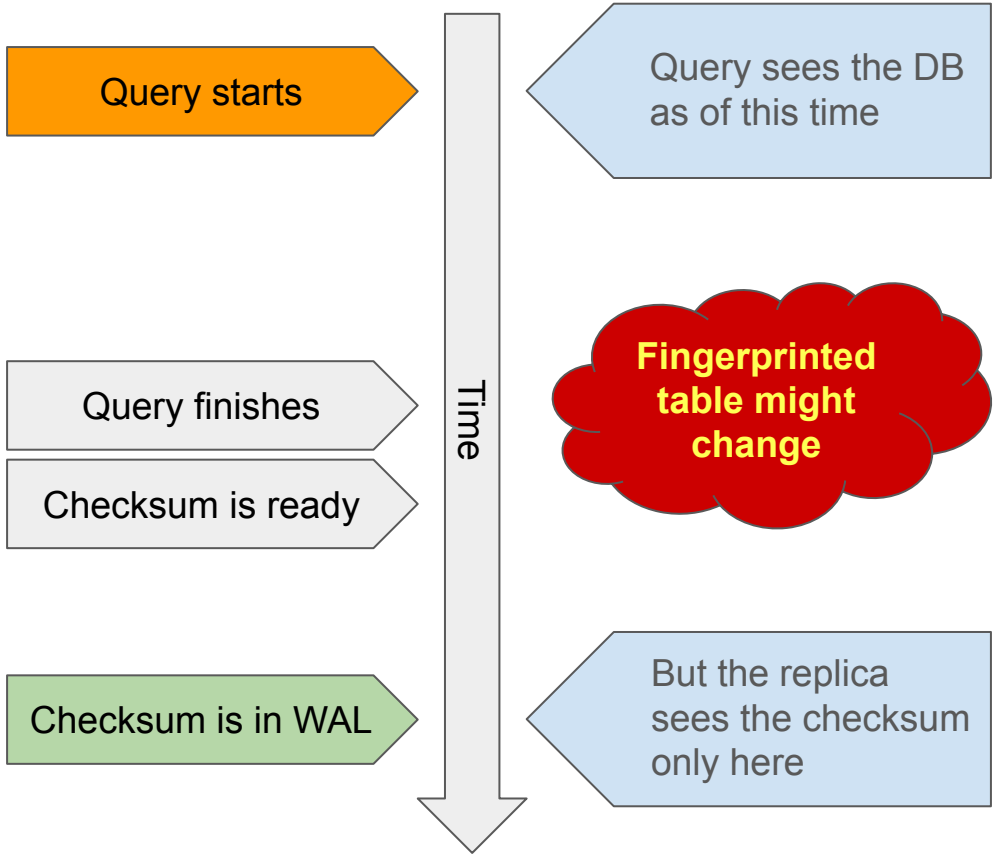
```
bench=# SELECT pg_logical_emit_message(true, 'w2jfp', to_jsonb(t)::text)
bench-# FROM query_fingerprint('SELECT * FROM pgbench_tellers', '1234') AS t;
 pg_logical_emit_message
-------------------------
 4/2A6CED0
(1 row)
```

Resulting message in the WAL:


```
{
  "action":"M",
  "xid":650960,
  "transactional":true,
  "prefix":"w2jfp",
  "content":"{\"fp\": \"63582df0152b44c2b5d83a645e93d235\",
             \"qiv\": \"b93501ddfd5b96fec808b55e953ed5a7\",
             \"qsig\": \"cc2adfa076a73371e326e9230d27bad8\",
             \"snap\": \"650960:650960:\",
             \"query\": \"SELECT * FROM pgbench_tellers\"}"
}
```

Fingerprint Snapshot

- Capture MVCC snapshot together with the checksum
- Delay replay to allow for the checksum to appear in the WAL
- Once a checksum message has been read, replay up to the specified MVCC snapshot, run the query and compare the result



Current status

Have

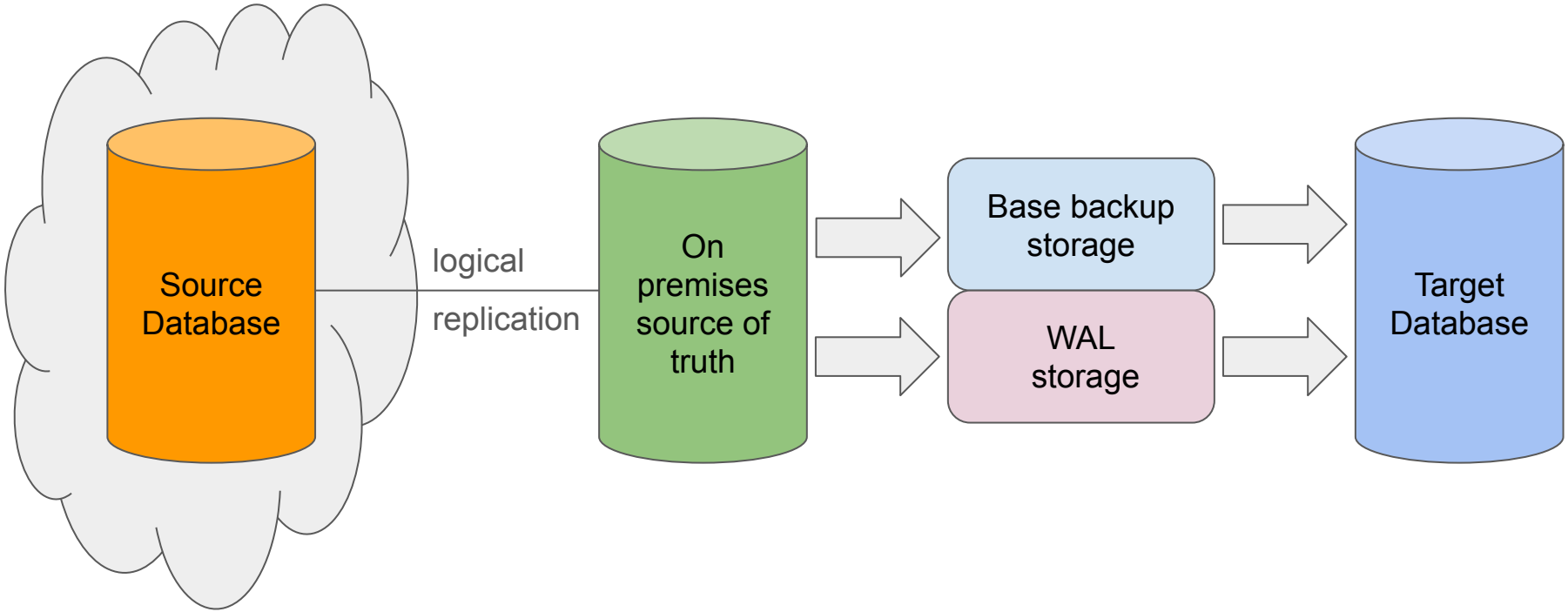
- in-house code
- in production
- sufficient for all our databases

Want

- rewrite as open source
- DDL handling + sequences
- use `fanotify` instead of `inotify`
- use MVCC snapshot as replication starting point instead of LSN
- use `pgoutput` either instead or as an option besides `wal2json`
- better storage format, perhaps Go's `msgpack`
- multi-thread replay (not sure if possible)
- transaction streaming (impossible with wal2json)
- more target systems, bigquery etc.
- filter and transformer

Alternative

This would have covered my initial problem



Talk to me if you found that interesting

Torsten Foertsch

tfoertsch123@gmail.com